International Standard

ISO/IEC 18181-1

Information technology — JPEG XL image coding system —

Part 1:
Core coding system

Technologies de l'information — Système de codage d'images JPEG XL —

Partie 1: Système de codage de noyau

Second edition
2024-07

Reference number
ISO/IEC 18181-1:2024(en)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 29, Coding of audio, picture, multimedia and hypermedia information.

This second edition cancels and replaces the first edition (ISO/IEC 18181-1:2022), which has been technically and editorially revised. It also incorporates the Amendment ISO/IEC 18181-1:2022/Amd 1:2022.

The main changes are as follows:

- technical corrections and clarifications, in particular to correct the values of various constants, correct errors in pseudocode, and clarify ambiguities in order to remove discrepancies between this document and ISO/IEC 18181-3 and ISO/IEC 18181-4;
- a thorough update of the document structure in order to improve clarity of presentation and to obtain a more logical ordering of the material from the point of view of decoder implementation.

A list of all parts in the ISO/IEC 18181 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Information technology — JPEG XL image coding system —

Part 1:
Core coding system

1 Scope

This document specifies a set of compression methods for coding one or more images of bi-level, continuous-tone greyscale, or continuous-tone colour, or multichannel digital samples.

This document:

— specifies decoding processes for converting compressed image data to reconstructed image data;

— specifies a codestream syntax containing information for interpreting the compressed image data;

— provides guidance on encoding processes for converting source image data to compressed image data.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO 15076-1, *Image technology colour management — Architecture, profile format and data structure — Part 2: Based on ICC.1:2022*

ISO/IEC 60559, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

IEC 61966-2-1, *Multimedia systems and equipment — Colour measurement and management — Part 2-1: Colour management — Default RGB colour space — sRGB*

ITU-R BT.2100-2, *Image parameter values for high dynamic range television for use in production and international programme exchange*

ITU-R BT.709-6, *Parameter values for the HDTV standards for production and international programme exchange*

IETF RFC 7932:2016, *Brotli Compressed Data Format*

SMPTE ST 428-1, *D-Cinema Distribution Master — Image Characteristics*

3 Terms, definitions and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org

3.1.1
bitstream
sequence of bytes

3.1.2
codestream
bitstream representing compressed image data

3.1.3
bundle
structured data consisting of one or more fields

3.1.4
field
numerical value or bundle, or an array of either

3.1.5
histogram
array of unsigned integers representing a probability distribution, used for entropy coding

3.2 Inputs

3.2.1
pixel
vector of dimension corresponding to the number of channels, consisting of samples

3.2.2
sample
integer or real value, of which there is one per channel per pixel

3.2.3
grid
2-dimensional array

3.2.4
sample grid
common coordinate system for all samples of an image, with top-left coordinates (0, 0), the first coordinate increasing towards the right, and the second increasing towards the bottom

3.2.5
channel component
rectangular array of samples having the same designation, regularly aligned along a sample grid

3.2.6
rectangle
rectangular area within a channel or grid

3.2.7
width
width in samples (number of columns) of a sample grid or a rectangle

3.2.8
height
height in samples (number of rows) of a sample grid or a rectangle

3.2.9
frame
single image (possibly part of an animation, composite, or multi-page image), i.e. a 2-dimensional array of pixels

3.2.10
greyscale
image representation in which each pixel is defined by a single sample representing intensity (either luminance or luma depending on the ICC profile)

3.2.11
opsin
photosensitive pigments in the human retina, having dynamics approximated by the XYB colour space

3.2.12
animation
series of pictures and timing delays to display as a video medium

3.2.13
tick
unit of time such that animation frame durations are integer multiples of the tick duration

3.2.14
composite
series of images that are superimposed

3.2.15
multi-page image
sequence of pictures to display in a paged way

3.2.16
preview
lower-fidelity rendition of one of the frames (e.g., lower resolution), or a frame that represents the entire content of all frames

3.3 Processes

3.3.1
decoding process
process which takes as its input a codestream and outputs an image

3.3.2
decoder
embodiment of a decoding process

3.3.3
encoding process
process which takes as its input image(s) and outputs compressed image data in the form of a codestream

3.3.4
encoder
embodiment of an encoding process

3.3.5
lossless
descriptive term for encoding and decoding processes in which the output of a decoding procedure is identical to the input to the encoding procedure

3.3.6
lossy
descriptive term for encoding and decoding processes which are not lossless

3.3.7
upsampling
procedure by which the (nominal) spatial resolution of a channel is increased

3.3.8
downsampling
procedure by which the spatial resolution of a channel is reduced

3.3.9
entropy encoding
lossless procedure designed to convert a sequence of input symbols into a sequence of bits such that the average number of bits per symbol approaches the entropy of the input symbols

3.3.10
entropy encoder
embodiment of an entropy encoding procedure

3.3.11
entropy decoding
lossless procedure which recovers the sequence of symbols from the sequence of bits produced by the entropy encoder

3.3.12
entropy decoder
embodiment of an entropy decoding procedure

3.3.13
channel decorrelation
method of reducing total encoded entropy by removing correlations between channels

3.3.14
channel correlation factor
factor by which a channel should be multiplied by before adding it to another channel to undo the channel decorrelation process

3.3.15
VarDCT
lossy encoding of a frame that applies DCT to varblocks

3.3.16
quantization
method of reducing the precision of (DCT) coefficients

3.3.17
quantization weight
factor that a quantized coefficient is multiplied by prior to application of the inverse DCT in the decoding process

3.4 Image and codestream organization

3.4.1
raster order
access pattern from left to right in the top row, then in the row below and so on

3.4.2
naturally aligned
positioning of a power-of-two sized rectangle such that its top and left coordinates are divisible by its width and height, respectively

3.4.3
block
naturally aligned square rectangle covering up to 8 × 8 input pixels

3.4.4
varblock
variable-size rectangle of one or more blocks

3.4.5
group
naturally aligned rectangle covering up to $2^n \times 2^n$ input pixels, with n between 7 and 10, inclusive

3.4.6
table of contents
data structure that enables seeking to a group or the next frame within a codestream

3.4.7
section
part of the codestream with an offset and length that are stored in a frame's table of contents

3.4.8
coefficient
input value to the inverse DCT

3.4.9
pass
data enabling decoding of successively more detail

3.4.10
LF group
LF values from a naturally aligned rectangle covering up to $2^{n+3} \times 2^{n+3}$ input pixels

3.5 Abbreviated terms

DCT	discrete cosine transform (as specified in I.7)
HF	all DCT coefficients apart from the LF coefficients, i.e. the high frequency coefficients
IDCT	inverse discrete cosine transform (as specified in I.7)
LF	lowest frequency coefficients of DCT coefficients, i.e. block averages
RGB	additive colour model with red, green, blue channels
XYB	absolute colour space based on gamma-corrected LMS (a colour space representing the response of cone cells in the human eye), in which X is derived from the difference between L and M (red-green), Y is an average of L and M (behaves similarly to luminance), and B is derived from S (blue)

4 Conventions

4.1 Mathematical symbols

[a, b], (c, d), [e, f)	closed or open or half-open intervals containing all integers or real numbers x (depending on context) such that $a \leq x \leq b$, $c < x < d$, $e \leq x < f$
{a, b, c}	ordered sequence of elements
`pi`	the smallest positive zero of the sine function (π)

4.2 Functions

`pow(x,y)`	exponentiation, `x` to the power of `y`.
`sqrt(x)`	square root, such that `pow(sqrt(x),2) == x` and `sqrt(x) >= 0`. Undefined for `x < 0`.
`cbrt(x)`	cube root, such that `pow(cbrt(x),3) == x`.
`cos(r)`	cosine of the angle `r` (in radians).
`erf(x)`	Gauss error function: `erf(x)` $= \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$
`log(x)`	natural logarithm of `x`. Undefined for `x <= 0`.
`log2(x)`	base-two logarithm of `x`. Undefined for `x <= 0`.
`floor(x)`	the largest integer that is less than or equal to `x`.
`ceil(x)`	the smallest integer that is greater than or equal to `x`.
`abs(x)`	absolute value of `x`: equal to `-x` if `x < 0`, otherwise `x`.
`sign(x)`	sign of `x`, 0 if `x` is 0, +1 if `x` is positive, −1 if `x` is negative.
`len(a)`	length (number of elements) of array `a`.
`sum(a)`	sum of all elements of the array/tuple/sequence `a`.
`max(a)`	maximal element of the array/tuple/sequence `a`.
`min(a)`	smallest element of the array/tuple/sequence `a`.
`UnpackSigned(u)`	equivalent to `u/2` if `u` is even, and `-(u + 1)/2` if `u` is odd.
`clamp(x, lo, hi)`	equivalent to `min(max(lo, x), hi)`.
`InterpretAsF16(u)`	the real number resulting from interpreting the unsigned 16-bit integer `u` as a binary16 floating-point number representation (cf. ISO/IEC 60559)
`InterpretAsF32(u)`	the real number resulting from interpreting the unsigned 32-bit integer `u` as a binary32 floating-point number representation (cf. ISO/IEC 60559)
`NarrowToI32(x)`	the signed 32-bit integer corresponding to the 32 least significant bits of the two's complement representation of `x`
`BitSet(u,b)`	the bit corresponding to `b` is 1 in `u`: true if and only if `u & b` (bitwise AND)

4.3 Operators

This document uses the operators defined by the C++ programming language (ISO/IEC 14882), with the following differences:

`/`	division of real numbers without truncation or rounding. Division by zero is undefined.
`<<`	left shift: `x << s` is defined as `x * pow(2,s)`
`>>`	right shift: `x >> s` is defined as `floor(x / pow(2,s))`
`Umod`	remainder: `a Umod d` is the unique integer `r` in `[0, d)` for which `a == r + q * d` for a suitable integer `q`

`Idiv`	integer division: `a Idiv b` is equivalent to `a / b`, rounded towards zero to an integer value
`and`	logical AND (`&&` in C++), true if and only if both operands are true, with short-circuit evaluation
`or`	logical OR (`||` in C++), false if and only if both operands are false, with short-circuit evaluation

The order of precedence for these operators is listed in Table 1 in descending order. If several operators appear in the same line, they have equal precedence. When several operators of equal precedence appear at the same level in an expression, evaluation proceeds according to the associativity of the operator (either from right to left or from left to right).

Table 1 — Operator precedence

Operators	Type of operation	Associativity
`++x`, `- -x`	prefix increment/decrement	left to right
`x++`, `x- -`	postfix increment/decrement	left to right
`!`, `~`	logical/bitwise NOT	right to left
`*`, `/`, `Idiv`, `Umod`	multiplication, division, integer division, remainder	left to right
`+`, `-`	addition and subtraction	left to right
`<<`, `>>`	left shift and right shift	left to right
`<` , `>`, `<=`, `>=`, `==`, `!=`	relational	left to right
`&`	bitwise AND	left to right
`^`	bitwise XOR	left to right
`|`	bitwise OR	left to right
`and`	logical AND	left to right
`or`	logical OR	left to right
`=`	assignment	right to left
`+=`, `-=`, `*=`	compound assignment	right to left

4.4 Pseudocode

This document describes functionality using pseudocode formatted as follows:

```
// Informative comment
var = u(8);
if (var == 1) return;  // Stop executing this code snippet
/* Normative specification: var != 0 */
(out1, out2) = Function(var, kConstant);
```

Variables such as `var` are typically referenced by text outside the source code.

The semantics of this pseudocode are those of the C++ programming language (ISO/IEC 14882), except that:

— Symbols from 4.1 and functions from 4.2 are allowed;

— Division, remainder and exponentiation are expressed as specified in 4.3;

— Functions can return tuples which unpack to variables as in the above example;

— `/* */` enclose normative directives specified using prose;

— All integers are stored using two's complement;

— Expressions and variables of which types are omitted, are understood as real numbers.

Where unsigned integer wraparound and truncated division are required, `Umod` and `Idiv` (see 4.3) are used for those purposes.

5 Functional concepts

5.1 Image organization

A channel is defined as a rectangular array of (integer or real) samples regularly aligned along a sample grid of `width` sample positions horizontally and `height` sample positions vertically. The number of channels may be 1 to 4099 (see `num_extra` in D.3).

A pixel is defined as a vector of dimension corresponding to the number of channels, consisting of samples with a position matching that of the pixel.

An image is defined as a two-dimensional array of pixels, and its width is `width` and height is `height`. Unless otherwise mentioned, `raster order' is used: left to right column within the topmost row, then left to right column within the row below the top, and so on until the rightmost column of the bottom row.

A codestream may contain multiple image frames. These can constitute an animation (a sequence of images) or a composite still image. Frames can have dimensions that differ from the image dimensions, and they can be blended over preceding frames. The frame that is being decoded is referred to as the `current' frame.

5.2 Mirroring

Some operations access samples with coordinates `cx`, `cy` that are outside the current frame bounds. The decoder redirects such accesses to a valid sample at the coordinates `Mirror(cx, cy)` as defined in the following code:

```
Mirror1D(coord, size) {
  if (coord < 0) return Mirror1D(-coord - 1, size);
  else if (coord >= size) return Mirror1D(2 * size - 1 - coord, size);
  else return coord;
}

Mirror(x, y) {
  return (Mirror1D(x, width), Mirror1D(y, height));
}
```

5.3 Group splitting

Channels are logically partitioned into naturally-aligned groups of `group_dim` × `group_dim` samples. The effective dimension of a group (i.e. how many pixels to read) can be smaller than `group_dim` for groups on the right or bottom of the image. The decoder ensures the decoded image has the dimensions specified in SizeHeader (D.2) by cropping at the right and bottom as necessary. Unless otherwise specified, `group_dim` is 256.

LF groups likewise consist of `group_dim` × `group_dim` LF samples, with the possibility of a smaller effective size on the right and bottom of the image.

Groups can be decoded independently. A 'table of contents' (F.3) stores the size (in bytes) of each group to allow seeking to any group. An optional permutation allows arbitrary reordering of groups within the codestream.

5.4 Codestream organization

A bitstream is a sequence of bytes. A codestream is a bitstream that represent compressed image data and metadata. N bytes can also be viewed as 8 × N bits. The first 8 bits are the bits constituting the first byte, in least to most significant order, the next eight bits (again in least to most significant order) constitute the second byte, and so on. Unless otherwise specified, bits are read from the codestream as specified in B.2.1.

NOTE Ordering bits from least to most significant allows using special CPU instructions to isolate the least-significant bits.

6 Encoder requirements

An encoder is an embodiment of the encoding process. This document does not specify an encoding process, and any encoding process is acceptable as long as the codestream conforms to the codestream syntax specified in this document. Annex O provides an informative description of such an encoding process.

7 Decoder requirements

A decoder is an embodiment of the decoding process. The decoder reconstructs sample values (arranged on a rectangular sampling grid) from a codestream as specified in this document. Annexes A to N define an output that alternative implementations shall duplicate.

Annex A
(normative)

Codestream overview

A.1 Codestream structure

The codestream consists of image headers (Annexes D and N) and an ICC profile, if present (E.4), followed by one or more frames (Annex F), as shown in Table A.1. This table and those it references, together with the syntax description (B.1), specify how a decoder reads the headers and frame(s).

Table A.1 — Codestream structure

condition	type	name	Annex
	Headers	`headers`	Annex D, Annex N
`headers.metadata.colour_encoding.want_icc`		`icc`	E.4
`headers.metadata.have_preview`	Frame	`preview_frame`	Annex F
	Frame	`frames[0]`	Annex F
`for (i = 1; !frames[i - 1].frame_header.is_last; ++i)`	Frame	`frames[i]`	Annex F

The `preview_frame` is a self-contained frame whose FrameHeader (F.2) specifies `lf_level=0`, `frame_type=kRegularFrame` and `save_as_reference=0`.

A.2 Decoding process

Annex B specifies how the decoder parses header information.

Annex C specifies how the decoder reads entropy-coded data.

After reading the image header (Annexes D and E) and the frame header (Annex F), the decoder reads the frame data (Annex G). The reconstruction of the decoded image can be viewed as a pipeline with multiple stages.

Some Annexes or subclauses begin with a condition; the decoder only applies the processing steps described in such an Annex or subclause if its condition holds true.

Annex H specifies the **kModular** frame encoding and Modular sub-bitstreams.

Annex I specifies the **kVarDCT** frame encoding.

The decoder applies restoration filters as specified in Annex J.

The presence/absence of additional image features (patches, splines and noise) is indicated in the frame header. The decoder draws these as specified in Annex K. Image features (if present) are rendered after restoration filters (if enabled), in the listed order.

Finally, the decoder performs colour transforms as specified in Annex L.

NOTE For an introduction to the coding tools and additional background, please refer to Reference [3].

Annex B
(normative)

Header syntax

B.1 General

The codestream is organized into "bundles" consisting of one or more conceptually related "fields". A field can be a bundle, or an array/value of a type from B.2. The graph of contained-within relationships between types of bundles is acyclic. This document specifies the structure of each bundle using tables structured as in Table B.1, with one row per field (in top to bottom order).

Table B.1 — Example structure of a table describing a bundle

condition	type	default	name
	Bool()	`false`	`div8`
`div8`	1 + u(5)	0	`h_div8`
`!div8`	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	`8 * h_div8`	`height`
	u(3)	0	`ratio`
`div8 and !ratio`	1 + u(5)	0	`w_div8`
`!div8 and !ratio`	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	`d_width`	`width`

If the condition is blank or evaluates to `true`, the field with the name in the **name** column is read from the codestream (B.1.1) according to the field type in the **type** column. Otherwise, the field is instead initialized to default value in the **default** column (B.1.2).

A condition of the form `for(i = 0; condition; ++i)` is equivalent to replacing this row with a sequence of rows obtained from the current one by removing its condition and replacing the value of `i` in each column with the consecutive values assumed by the loop-variable `i` until `condition` is `false`. If `condition` is initially `false`, the current row has no effect.

If name ends with `[n]` then the field is a fixed-length array with n entries. If condition is blank or evaluates to true, each array element is read from the codestream (B.1.1) in order of increasing index. Otherwise, each element is initialized to default (B.1.2). The name is potentially referenced in the condition or type of a subsequent row, or the condition of the same row.

B.1.1 Reading a field

If a field is to be read from the codestream, the type entry determines how to do so. If it is a basic field type, it is read as described in B.2. Otherwise type is a (nested) bundle denoted Nested, residing within a parent bundle Parent. Nested is read as if the rows of the table defining Nested were inserted into the table defining Parent in place of the row that defined Nested. This principle is applied recursively, corresponding to a depth-first traversal of all fields.

B.1.2 Initializing a field

If a field is to be initialized to default, the type entry determines how to do so. If it is a bundle, then default is blank and each field of the bundle is (recursively) initialized to the default specified within the bundle's table. Otherwise, the field is set to default, which is a valid value of the same type.

B.2 Field types

B.2.1 u(n)

u(0) evaluates to the value zero without reading any bits. For n > 0, u(n) reads n bits from the codestream, advances the position accordingly, and returns the value in the range $[0, 2^n)$ represented by the bits. The decoder first reads the least-significant bit of the value, from the least-significant not yet consumed bit in the first not yet fully consumed byte of the codestream. The next bit of the value (in increasing order of significance) is read from the next (more significant) bit of the same byte unless all its bits have already been consumed, in which case the decoder reads from the least-significant bit of the next byte, and so on.

B.2.2 U32(d0, d1, d2, d3)

In this subclause, "distribution" refers to one of the following three encodings of a range of values: `u`, u(n), or offset + u(n). The d0, d1, d2, d3 parameters represent distributions.

U32(d0, d1, d2, d3) reads an unsigned 32-bit integer `value` in $[0, 2^{32})$ as follows. The decoder first reads a u(2) from the codestream indicating which distribution to use (0 selects d0, 2 selects d2 etc.). Let d denote this distribution, which determines how to decode `value`.

If d is `u`, `value` is the integer `u`. If d is u(n), value is read as u(n). If d is offset + u(n), the decoder reads `v = u(n)`. The resulting value is `(offset + v) Umod (1 << 32)`.

NOTE The value of `u` is implicitly defined and not stored explicitly in the codestream.

EXAMPLE For a field of type U32(8, 16, 32, u(7)), the bits 10 result in `value = 32`. For a U32(u(2), u(4), u(6), u(8)) field, the bits 010111 result in `value = 7`.

B.2.3 U64()

U64() reads an unsigned 64-bit integer `value` in $[0, 2^{64})$ using a single variable-length encoding. The decoder first reads a u(2) selector `s`. If `s == 0`, `value` is 0. If `s == 1`, value is 1 + u(4). If `s == 2`, value is 17 + u(8). Otherwise (`s == 3`), `value` is read as specified by the following code:

```
value = u(12); shift = 12;
while (u(1) == 1) {
  if (shift == 60) {
    value += u(4) << shift; // only 4, we already read 60
    break;
  }
  value += u(8) << shift; shift += 8;
}
```

EXAMPLE The largest possible value $2^{64} - 1$ is encoded as 73 consecutive 1-bits.

B.2.4 F16()

F16() reads a real value in [−65504, 65504], as specified by the following code:

```
bits16 = u(16);
biased_exp = (bits16 >> 10) & 0x1F;
value = InterpretAsF16(bits16);
```

The value of `biased_exp` is not 31.

NOTE This rules out NaN and infinities.

B.2.5 Bool()

Bool() reads a boolean value as `u(1) ? true : false`.

B.2.6 Enum(EnumTable)

Enum(EnumTable) reads v = U32(0, 1, 2 + u(4), 18 + u(6)). The value v does not exceed 63 and is defined by the table titled EnumTable. Such tables are structured according to Table B.2, with one row per unique value.

Table B.2 — Example structure of a table describing an enumerated type

name	value	meaning
kD65	1	CIE Standard Illuminant D65: 0.3127, 0.3290
kCustom	2	Custom white point stored in `colour_encoding.white`
kE	10	CIE Standard Illuminant E (equal-energy): 1/3, 1/3
kDCI	11	DCI-P3 from SMPTE ST 428-1: 0.314, 0.351

An enumerated type is interpreted as having the meaning of the row with the corresponding value. The value in the **name** column (or where ambiguous, EnumTable.name) is an identifier used to refer to a particular enumerated value, and begins with the letter **k**, e.g., **kRGB**.

B.2.7 ZeroPadToByte()

The decoder skips to the next byte boundary if not already at a byte boundary. All skipped bits have value 0.

NOTE Since any padding bits are zero, they can serve as an additional indicator of codestream integrity.

B.3 Extensions

This bundle, specified in Table B.3, is a field in bundles (ImageMetadata, FrameHeader, RestorationFilter) which can be extended (cf. Annex N).

Table B.3 — Extensions bundle

condition	type	default	name
	U64()	0	`extensions`
`extensions != 0`	U64()	0	`extension_bits[NumExt]`

An extension consists of zero or more bits whose interpretation is established by Annex N. Each extension is identified by an `ext_id` in the range [0, 63), assigned in increasing sequential order. `extensions` is a bit array where the i-th bit (least-significant == 0) indicates whether the extension with `ext_id = i` is present. `NumExt` denotes the number of extensions present, i.e. number of 1-bits in `extensions`. Extensions that are present are stored in ascending order of their `ext_id`. `extension_bits[i]` indicates the number of bits stored for the extension whose `ext_id` = i, starting after `extension_bits` has been read. The decoder reads all these bits for all extensions which are present.

Annex C
(normative)

Entropy decoding

C.1 Overview

This Annex specifies an entropy decoder. Other Annexes and their subclauses indicate how it is used for various elements of the codestream.

The codestream contains multiple independently entropy-coded streams. During decoding of such a stream, the decoder maintains an entropy coder state, which is initialized as specified in C.2. This state has to persist and can be updated during the decoding of the entropy-coded stream, as specified in C.3.

The following variables (which are specified in the following clauses) are part of the entropy decoder state:

— `num_dist`, the pre-clustering number of distributions (context identifiers), which is given;

— `num_clusters`, the post-clustering number of distributions;

— `lz77`, a bundle with LZ77 settings;

— `clusters[num_dist]`, describing the context clustering;

— `configs[num_clusters]`, hybrid integer configurations;

— `use_prefix_code`, a Boolean flag indicating if prefix coding or ANS is used;

— `lz_len_conf`, a hybrid integer configuration used to decode LZ77 lengths;

— `window[1 << 20]`, the LZ77 window of previously-decoded symbols (initialized to all zeroes);

— `num_to_copy`, `copy_pos`, and `num_decoded`: counters used for copying symbols from the LZ77 window;

— If `use_prefix_code`: a set of `num_clusters` post-clustered prefix codes;

— If `!use_prefix_code`, i.e. when the stream uses ANS:

  — `log_alphabet_size`, indicating the alphabet size of the entropy-coded ANS symbols;

  — a set of `num_clusters` post-clustered probability distributions `D[1 << log_alphabet_size]`;

  — the alias mapping for each post-clustered distribution (`log_bucket_size`, `bucket_size`, `symbols`, `offsets`, and `cutoffs`);

  — `state`, a 32-bit unsigned integer that represents the ANS state.

NOTE As an implementation optimization, the variables `lz_len_conf`, `window`, `num_to_copy`, `copy_pos`, and `num_decoded` can be ignored in case `lz77.enabled` is `false`.

C.2 Distribution decoding

C.2.1 General

This subclause describes how to decode the probability distributions and other stream initialization information that is needed before the actual entropy decoding can start. When this Annex is referenced, `num_dist` is given; it denotes the number of pre-clustered probability distributions.

The decoder first reads the LZ77 settings `lz77` as specified in Table C.1.

Table C.1 — LZ77Params bundle

condition	type	name
	Bool()	`enabled`
`enabled`	U32(224, 512, 4096, 8 + u(15))	`min_symbol`
`enabled`	U32(3, 4, 5 + u(2), 9 + u(8))	`min_length`

If `lz77.enabled`, the decoder sets `lz_dist_ctx = num_dist++`, and reads `lz_len_conf = HybridUintConfig(8)` as specified in C.2.3.

The decoder then reads the mapping `clusters` of the `num_dist` distributions into `num_clusters` clusters as specified in C.2.2. It then reads `use_prefix_code` as a Bool(). If `use_prefix_code` is `false`, the decoder sets `log_alphabet_size` to 5 + u(2); otherwise, it sets `log_alphabet_size` to 15. Then for each post-clustered distribution, in increasing order of index `i` in [0, `num_clusters`), the decoder reads `configs[i] = HybridUintConfig(log_alphabet_size)` as specified in C.2.3.

The decoder then reads `num_clusters` post-clustered distributions `D[i]` as follows. If `use_prefix_code` is `true`, then for `i` in [0, `num_clusters`): if Bool() is `false`, `count[i] = 1`, otherwise `n` = u(4) is read, and `count[i] = 1 + (1 << n) + u(n)`, which is at most `1 << 15`. After reading the counts, the decoder reads each `D[i]` (implicitly described by a prefix code) as specified in C.2.4, with `alphabet_size = count[i]`. If `use_prefix_code` is `false`, the distributions are read as specified in C.2.5.

C.2.2 Distribution clustering

The probability distributions that symbols belong to can be clustered together. This subclause specifies how this clustering is read by the decoder.

The output of this procedure is an array `clusters` with `num_dist` entries (one for each pre-clustered context), with values in the range [0, `num_clusters`). All integers in [0, `num_clusters`) are present in this array, and `num_clusters < 256`. Position i in this array indicates that context i is merged into the corresponding cluster.

If `num_dist == 1`, then `num_clusters = 1` and `clusters[0] = 0`, and the remainder of this subclause is skipped.

The decoder first reads a Bool() `is_simple` indicating a simple clustering. If `is_simple` is `true`, it then decodes a u(2) representing the number of bits per entry `nbits`. For each pre-clustered distribution, the decoder reads a u(`nbits`) value that indicates the cluster that the given distribution belongs to.

Otherwise, if `is_simple` is `false`, the decoder reads a Bool() `use_mtf`. The decoder then initializes a symbol decoder using a single distribution D, as specified in C.2.1, where if `num_dist == 2` then in the recursive distribution decoding, `lz77.enabled` is `false`. For each pre-clustered distribution i, the decoder reads an integer as specified in C.3.3. Finally, if `use_mtf`, the decoder applies an inverse move-to-front transform to the cluster mapping (see code below).

```
MTF(v[256], index) { value = v[index];
                     for (i = index; i; --i) v[i] = v[i - 1];
                     v[0] = value; }
InverseMoveToFrontTransform(clusters[num_dist]) {
  for (i = 0; i < 256; ++i) mtf[i] = i;
  for (i = 0; i < num_dist; ++i) {
    index = clusters[i]; /* index < 256 */
    clusters[i] = mtf[index];
    if (index != 0) MTF(mtf, index);
  }
}
```

C.2.3 Hybrid integer configuration

The configuration for the hybrid unsigned integer decoder of C.3.3 is read from the bitstream as follows:

```
ReadUintConfig(log_alphabet_size) {
  config.split_exponent = u(ceil(log2(log_alphabet_size + 1)));
  if (config.split_exponent != log_alphabet_size) {
    nbits = ceil(log2(config.split_exponent + 1));
    config.msb_in_token = u(nbits);
    /* config.msb_in_token is <= config.split_exponent */
    nbits = ceil(log2(config.split_exponent - config.msb_in_token + 1));
    config.lsb_in_token = u(nbits);
  } else { config.msb_in_token = 0; config.lsb_in_token = 0; }
  /* config.lsb_in_token + config.msb_in_token is <= config.split_exponent */
  config.split = 1 << config.split_exponent;
  return config;
}
```

C.2.4 Histogram and prefix code

A Huffman histogram stream stores metadata required for decoding a Huffman symbol stream. See IETF RFC 7932:2016 sections 3.4 (“Simple Prefix Codes”) and 3.5 (“Complex Prefix Codes”) for format encoding description and definitions. The alphabet size mentioned in the RFC is explicitly specified as parameter `alphabet_size` when the histogram is being decoded, except in the special case of `alphabet_size == 1`, where no histogram is read, and all decoded symbols are zero without reading any bits at all.

A Huffman symbol stream stores a sequence of unsigned integers (symbols). To read a symbol, the decoder reads a series of bits via u(1), until the bits concatenated in left-to-right order exactly match one of the prefix codes associated with the symbol. See IETF RFC 7932:2016 section 3.2 (“Use of Prefix Coding in the Brotli Format”) for a detailed description of how to build the prefix codes from a Huffman histogram.

C.2.5 ANS distribution decoding

ANS probabilities distributions are represented as an array with `1 << log_alphabet_size` elements with values in [0, `1 << 12`], one per symbol, that sum to `1 << 12`, i.e. they can be interpreted as representing symbol probabilities with an implicit denominator of `1 << 12`. Indexing the array with an out-of-bounds value results in a 0 value.

To decode a distribution D, the decoder follows the following procedure (where ’read a number according to a prefix code’ refers to C.2.4):

```
U8() { // reads an integer value in the range [0, 256) using [1, 12) bits
  if (u(1) == 0) return 0;
  else { n = u(3); return u(n) + (1 << n); }
}

table_size = 1 << log_alphabet_size;
/* initialize D as array with table_size entries with value 0 */
if (Bool()) {
  if (Bool()) { v1 = U8(); v2 = U8(); /* v1 != v2 */
                D[v1] = u(12); D[v2] = (1 << 12) - D[v1];
                alphabet_size = 1 + max(v1, v2);
  } else { x = U8(); D[x] = 1 << 12; alphabet_size = 1 + x; }
  return;
}
if (Bool()) {
  alphabet_size = U8() + 1;
  for (i = 0; i < alphabet_size; i++) D[i] = (1 << 12) Idiv alphabet_size;
  for (i = 0; i < ((1 << 12) Umod alphabet_size); i++) D[i]++;
  return;
}
len = 0; while (len < 3) { if (Bool()) len++; else break; }
shift = u(len) + (1 << len) - 1;
/* shift <= 13 */
alphabet_size = U8() + 3;

omit_log = -1; omit_pos = -1;
/* initialize same as array with alphabet_size entries with value 0 */
for (i = 0; i < alphabet_size; i++) {
  logcounts[i] = /* read a number according to the following prefix code:
    Number   Code       Number   Code (bits parsed from right to left)
    0        10001      7        010
    1         1011       8          101
    2         1111       9          110
    3         0011       10         000
    4         1001       11         100001
    5         0111       12         0000001
    6         100        13         1000001
  */
  if (logcounts[i] == 13) { rle = U8(); same[i] = rle + 5; i += rle + 3; continue; }
  if (logcounts[i] > omit_log) { omit_log = logcounts[i]; omit_pos = i; }
}
/* omit_pos >= 0 */
/* omit_pos + 1 >= alphabet_size || logcounts[omit_pos + 1] != 13 */
prev = 0; numsame = 0;
for (i = 0; i < alphabet_size; i++) {
  if (same[i] != 0) { numsame = same[i] - 1; prev = i > 0 ? D[i - 1] : 0; }
  if (numsame > 0) { D[i] = prev; numsame--; }
  else {
    code = logcounts[i];
    if (i == omit_pos || code == 0) continue;
    else if (code == 1) D[i] = 1;
    else { bitcount = min(max(0, shift - ((12 - code + 1) >> 1)), code - 1);
           D[i] = (1 << (code - 1)) + (u(bitcount) << (code - 1 - bitcount)); }
  }
  total_count += D[i];
}
D[omit_pos] = (1 << 12) - total_count;
/* D[omit_pos] >= 0 */
```

C.2.6 Alias mapping

For a given probability distribution D of symbols in the range [0, `1 << log_alphabet_size`), tables of `symbols`, `offsets` and `cutoffs` are initialized according to the code below.

```
log_bucket_size = 12 - log_alphabet_size;
bucket_size = 1 << log_bucket_size;
table_size = 1 << log_alphabet_size;
if (/* amount of symbols with nonzero probability in D */ == 1) {
  s = /* index of the symbol with nonzero probability */;
  for (i = 0; i < table_size; i++) {
    symbols.push_back(s); offsets.push_back(bucket_size * i); cutoffs.push_back(0);
  }
  return;
}
for (i = 0; i < alphabet_size; i++) {
  cutoffs.push_back(D[i]); symbols.push_back(i);
  if (cutoffs[i] > bucket_size) overfull.push_back(i);
  else if(cutoffs[i] < bucket_size) underfull.push_back(i);
}
for (i = alphabet_size; i < table_size; i++) {
  cutoffs[i] = 0;
  underfull.push_back(i);
}
while (overfull.size() > 0) {
  /* underfull.size() > 0 */
  o = overfull.back(); u = underfull.back(); overfull.pop_back(); underfull.pop_back();
  by = bucket_size - cutoffs[u];
  cutoffs[o] -= by; symbols[u] = o; offsets[u] = cutoffs[o];
  if (cutoffs[o] < bucket_size) underfull.push_back(o);
  else if (cutoffs[o] > bucket_size) overfull.push_back(o);
}
for (i = 0; i < table_size; i++) {
  if (cutoffs[i] == bucket_size) { symbols[i] = i; offsets[i] = 0; cutoffs[i] = 0; }
  else offsets[i] -= cutoffs[i];
}
```

The *alias mapping* of x under D and its corresponding initialized `symbols`, `offsets` and `cutoffs` is the output of the procedure `AliasMapping(x)` specified in the code below, that produces a `symbol` in [0, `1 << log_alphabet_size`) and an `offset` in [0, `1 << 12`).

```
AliasMapping(x) {
  i = x >> log_bucket_size; pos = x & (bucket_size - 1);
  symbol = pos >= cutoffs[i] ? symbols[i] : i;
  offset = pos >= cutoffs[i] ? offsets[i] + pos : pos;
  return (symbol, offset);
}
```

C.3 Symbol decoding

C.3.1 General

The decoder reads integer symbols from an entropy coded stream as specified in C.3.3. If `use_prefix_code`, entropy coded symbols are read according to C.2.4, using the corresponding prefix code. Otherwise, they are read according to C.3.2. In both cases the final decoded integers are reconstructed using a combination of context-dependent entropy decoding (using a prefix code or ANS), raw bits (read directly from the same stream), and possibly LZ77 (copying of previously decoded symbol sequences).

C.3.2 ANS symbol decoding

The ANS decoder keeps an internal 32-bit unsigned integer `state`. Upon initialization of a new ANS stream, the initial value of `state` is read as a u(32) from the codestream. At the end of an ANS stream, `state` is `0x130000`. There is only a single `state` variable per ANS stream; it is shared between different contexts.

The decoder decodes a symbol belonging to a given distribution D as specified by the code below:

```
index = state & 0xFFF;
(symbol, offset) = D.AliasMapping(index);
state = D[symbol] * (state >> 12) + offset;
if (state < (1 << 16)) state = (state << 16) | u(16);
```

C.3.3 Hybrid integer decoding

An unsigned integer is read from an entropy coded stream given a pre-clustered context identifier `ctx` as specified by `DecodeHybridVarLenUint` in the following code. `num_to_copy` is initialized to 0 and the array `window[]` is initialized to all zeroes when the first symbol is decoded from the stream. `dist_multiplier` is 0 unless otherwise specified.

```
ReadUint(config, token) {
  if (token < config.split) return token;
  n = config.split_exponent - config.msb_in_token - config.lsb_in_token +
      ((token - config.split) >> (config.msb_in_token + config.lsb_in_token));
  /* n < 32 */
  low = token & ((1 << config.lsb_in_token) - 1);
  token = token >> config.lsb_in_token;
  token &= (1 << config.msb_in_token) - 1;
  token |= (1 << config.msb_in_token);
  result = (((token << n) | u(n)) << config.lsb_in_token ) | low;
  /* result < (1 << 32) */
  return result;
}
kSpecialDistances[120][2] = {
  {0, 1}, {1, 0},  {1, 1}, {-1, 1}, {0, 2}, {2, 0}, {1, 2}, {-1, 2}, {2, 1}, {-2, 1}, {2, 2},
  {-2, 2}, {0, 3}, {3, 0}, {1, 3}, {-1, 3}, {3, 1}, {-3, 1}, {2, 3}, {-2, 3}, {3, 2},
  {-3, 2}, {0, 4}, {4, 0}, {1, 4}, {-1, 4}, {4, 1}, {-4, 1}, {3, 3}, {-3, 3}, {2, 4},
  {-2, 4}, {4, 2}, {-4, 2}, {0, 5}, {3, 4},  {-3, 4}, {4, 3}, {-4, 3}, {5, 0}, {1, 5},
  {-1, 5}, {5, 1}, {-5, 1}, {2, 5}, {-2, 5}, {5, 2}, {-5, 2}, {4, 4}, {-4, 4}, {3, 5},
  {-3, 5}, {5, 3}, {-5, 3}, {0, 6}, {6, 0}, {1, 6}, {-1, 6}, {6, 1}, {-6, 1}, {2, 6},
  {-2, 6}, {6, 2}, {-6, 2}, {4, 5}, {-4, 5}, {5, 4}, {-5, 4}, {3, 6}, {-3, 6}, {6, 3},
  {-6, 3}, {0, 7}, {7, 0},  {1, 7}, {-1, 7}, {5, 5}, {-5, 5}, {7, 1}, {-7, 1}, {4, 6},
  {-4, 6}, {6, 4}, {-6, 4}, {2, 7}, {-2, 7}, {7, 2}, {-7, 2}, {3, 7}, {-3, 7}, {7, 3},
  {-7, 3}, {5, 6}, {-5, 6}, {6, 5}, {-6, 5}, {8, 0}, {4, 7},  {-4, 7}, {7, 4}, {-7, 4},
  {8, 1}, {8, 2}, {6, 6}, {-6, 6}, {8, 3}, {5, 7}, {-5, 7}, {7, 5}, {-7, 5}, {8, 4}, {6, 7},
  {-6, 7}, {7, 6}, {-7, 6}, {8, 5}, {7, 7}, {-7, 7}, {8, 6}, {8, 7} };

DecodeHybridVarLenUint(ctx) {
  if (num_to_copy > 0) { r = window[(copy_pos++) & 0xFFFFF]; num_to_copy--; }
  else {
    token = /* Read symbol from entropy coded stream using D[clusters[ctx]] */;
    if (lz77.enabled and token >= lz77.min_symbol) {
      num_to_copy = ReadUint(lz_len_conf, token - lz77.min_symbol) + lz77.min_length;
      token = /* Read symbol using D[clusters[lz_dist_ctx]] */;
      distance = ReadUint(configs[clusters[lz_dist_ctx]], token);
      if (dist_multiplier == 0) distance++;
      else if (distance < 120) {
        offset = kSpecialDistances[distance][0];
        distance = offset + dist_multiplier * kSpecialDistances[distance][1];
        if (distance < 1) distance = 1;
      } else distance -= 119;
      distance = min(distance, num_decoded, 1 << 20);
      copy_pos = num_decoded - distance;
      return DecodeHybridVarLenUint(clusters[ctx]);
    }
    r = ReadUint(configs[clusters[ctx]], token);
  }
  window[(num_decoded++) & 0xFFFFF] = r;
  return r;
}
```

NOTE It is not necessarily the case that `num_to_copy` is zero at the end of the stream.

Annex D
(normative)

Image header

D.1 General

The decoder reads the Headers bundle (Table D.1) as specified in B.1.

Table D.1 — Headers bundle

condition	type	name	subclause
	u(16)	`signature`	
	SizeHeader	`size`	D.2
	ImageMetadata	`metadata`	D.3

The `signature` consists of the fixed two-byte sequence `0xFF0A`, i.e. it is equal to the value 2815.

D.2 Image dimensions

Table D.2 specifies the SizeHeader bundle.

Table D.2 — SizeHeader bundle

condition	type	default	name
	Bool()	`false`	`div8`
`div8`	1 + u(5)	0	`h_div8`
`!div8`	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	`8 * h_div8`	`height`
	u(3)	0	`ratio`
`div8 and !ratio`	1 + u(5)	0	`w_div8`
`!div8 and !ratio`	U32(1 + u(9), 1 + u(13), 1 + u(18), 1 + u(30))	`d_width`	`width`

The default value `d_width` is defined as `(ratio == 0 ? 8 * w_div8 : AspectRatio(height, ratio))`, where `AspectRatio` is computed as follows:

```
AspectRatio(height, ratio) {
  if (ratio == 1) return height;
  if (ratio == 2) return height * 6 Idiv 5;
  if (ratio == 3) return height * 4 Idiv 3;
  if (ratio == 4) return height * 3 Idiv 2;
  if (ratio == 5) return height * 16 Idiv 9;
  if (ratio == 6) return height * 5 Idiv 4;
  if (ratio == 7) return height * 2;
}
```

NOTE This encoding cannot represent a zero-width or zero-height image.

D.3 Image metadata

D.3.1 ImageMetadata

The ImageMetadata bundle, defined in Table D.3, holds information that applies to all Frames (including the preview); decoders use it to interpret pixel values and restore the original image.

Table D.3 — ImageMetadata bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	Bool()	false	extra_fields
extra_fields	1 + u(3)	1	orientation
extra_fields	Bool()	false	have_intr_size
have_intr_size	SizeHeader		intrinsic_size
extra_fields	Bool()	false	have_preview
have_preview	PreviewHeader		preview
extra_fields	Bool()	false	have_animation
have_animation	AnimationHeader		animation
!all_default	BitDepth		bit_depth
!all_default	Bool()	true	modular_16bit_buffers
!all_default	U32(0, 1, 2 + u(4), 1 + u(12))	0	num_extra
!all_default	ExtraChannelInfo		ec_info[num_extra]
!all_default	Bool()	true	xyb_encoded
!all_default	ColourEncoding		colour_encoding
extra_fields	ToneMapping		tone_mapping
!all_default	Extensions		extensions
	Bool()		default_m
!default_m and xyb_encoded	OpsinInverseMatrix		opsin_inverse_matrix
!default_m	u(3)	0	cw_mask
BitSet(cw_mask, 1)	F16()	d_up2	up2_weight[15]
BitSet(cw_mask, 2)	F16()	d_up4	up4_weight[55]
BitSet(cw_mask, 4)	F16()	d_up8	up8_weight[210]

If have_intr_size, then intrinsic_size denotes the recommended dimensions for displaying the image, i.e. applications are advised to resample the decoded image to the dimensions indicated in intrinsic_size.

have_preview indicates whether the codestream includes a preview_frame.

have_animation indicates whether an AnimationHeader is included (animation) and whether FrameHeader (F.2) includes timing information.

modular_16bit_buffers indicates whether signed 16-bit integers have a large enough range to store and apply inverse transforms to all the decoded samples in modular image sub-bitstreams (see Annex H). If false, 32-bit integers have to be used. In any case, signed 32-bit integers suffice to store decoded samples in modular image sub-bitstreams, as well as the results of inverse modular transforms. For some of the intermediate arithmetic (e.g., predictor computations), 64-bit arithmetic is needed.

num_extra is the number of additional image channels. If it is nonzero, then ec_info signals the semantics of each extra channel.

xyb_encoded is true if the stored image is in the XYB colour space. In L.2 it is specified how to convert XYB to RGB.

In the rest of this document, the three main colour components are referred to as X, Y, B, even in the case where the XYB colour space is not used for the stored image; in that case the actual components are either R, G, B or Cb, Y, Cr or C, M, Y.

colour_encoding indicates the colour image encoding of the original image. This may differ from the encoding of the stored image if xyb_encoded is true. This bundle is specified in Annex E.

tone_mapping (Table E.9) has information for mapping HDR images to lower dynamic range displays. This bundle is specified in E.3.

OpsinInverseMatrix is defined in L.2.1.

For the upsampling filters, custom weights are signalled depending on `cw_mask`. The default weights `d_up2`, `d_up4` and `d_up8` are defined in K.2.

D.3.2 Orientation

The `orientation` indicates which orientation transform the decoder applies after decoding the image (Table D.4), including the preview frame if present.

NOTE `orientation` matches the values used by JEITA CP-3451C (Exif version 2.3).

Table D.4 — Orientation

`orientation`	**first row**	**first column**	**transform to apply**
1	top	left	none
2	top	right	flip horizontally
3	bottom	right	rotate 180
4	bottom	left	flip vertically
5	left	top	transpose (rotate 90 clockwise then flip horizontally)
6	right	top	rotate 90 clockwise
7	right	bottom	flip horizontally then rotate 90 clockwise
8	left	bottom	rotate 90 counterclockwise

D.3.3 PreviewHeader

Table D.5 specifies the PreviewHeader bundle.

Table D.5 — PreviewHeader bundle

condition	type	default	name
	Bool()		`div8`
`div8`	U32(16, 32, 1 + u(5), 33 + u(9))	1	`h_div8`
`!div8`	U32(1 + u(6), 65 + u(8), 321 + u(10), 1345 + u(12))	`8 * h_div8`	`height`
	u(3)		`ratio`
`div8 and !ratio`	U32(16, 32, 1 + u(5), 33 + u(9))	1	`w_div8`
`!div8 and !ratio`	U32(1 + u(6), 65 + u(8), 321 + u(10), 1345 + u(12))	`d_width`	`width`

The default value `d_width` is defined as in SizeHeader: `d_width = (ratio == 0 ? 8 * w_div8 : AspectRatio(height, ratio))`.

The preview `width` and `height` do not exceed 4096.

D.3.4 AnimationHeader

AnimationHeader, defined in Table D.6, holds meta-information about an animation or image sequence that is present if `metadata.have_animation`.

Table D.6 — AnimationHeader bundle

condition	type	default	name
	U32(100, 1000, 1 + u(10), 1 + u(30))	0	`tps_numerator`
	U32(1, 1001, 1 + u(8), 1 + u(10))	0	`tps_denominator`
	U32(0, u(3), u(16), u(32))	0	`num_loops`
	Bool()	`false`	`have_timecodes`

`tps_numerator / tps_denominator` indicates the number of ticks per second.

`num_loops` is the number of times to repeat the animation (0 is interpreted as infinity).

`have_timecodes` indicates whether time codes are signalled in the frame headers.

NOTE This document defines the interval between presenting the current and next frame in units of ticks. In the case where `metadata.have_animation` is `false`, the frames do not represent an animation, but layers that are overlaid to obtain a composite image.

D.3.5 BitDepth

Table D.7 specifies the BitDepth bundle.

Table D.7 — BitDepth bundle

condition	type	default	name
	Bool()	`false`	`float_sample`
`!float_sample`	U32(8, 10, 12, 1 + u(6))	8	`bits_per_sample`
`float_sample`	U32(32, 16, 24, 1 + u(6))	8	`bits_per_sample`
`float_sample`	1 + u(4)	0	`exp_bits`

This information describes how to interpret Modular-encoded integer samples, for extra channels and for the colour channels if `!metadata.xyb_encoded`, as specified in G.4.2. For colour channels encoded with VarDCT and/or with `metadata.xyb_encoded`, this information is merely a suggestion for a sample representation that can be suitable for the image, but it does not impact the actual sample values.

`float_sample` is the data type of sample values (`true` for floating point, `false` for integers).

`bits_per_sample` is the number of bits per channel of the original image. The encoding of this value depends on `float_sample`: for integers the value is in range [1, 31], for floating point the value is in range [5, 32].

If `float_sample` is `false`, then integer samples of the decoded image are divided by `(1 << bits_per_sample) - 1`.

If `float_sample` is `true`, then `exp_bits` is the number of bits used for the exponent of floating point sample values. Let `mantissa_bits = bits_per_sample - exp_bits - 1`. Integer samples of the decoded image are interpreted as floating point values with 1 sign bit, the indicated amount of exponent bits and mantissa bits and otherwise following the principles of ISO/IEC 60559 with an exponent bias of `(1 << (exp_bits - 1)) - 1`. The value of `exp_bits` is in range [2, 8], and the value of `mantissa_bits` is in range [2, 23]. If a sample's exponent has the highest representable value, the decoded value of this sample is unspecified.

NOTE On some CPUs and floating-point formats, infinity/Not A Number are disallowed and the largest possible exponent is treated in the same way as the next smaller one.

D.3.6 ExtraChannelInfo

Table D.8 specifies the ExtraChannelInfo bundle.

Table D.8 — ExtraChannelInfo bundle

condition	type	default	name
	Bool()	`true`	`d_alpha`
`!d_alpha`	Enum(ExtraChannelType)	**kAlpha**	`type`
`!d_alpha`	BitDepth		`bit_depth`
`!d_alpha`	U32(0, 3, 4, 1 + u(3))	0	`dim_shift`
`!d_alpha`	U32(0, u(4), 16 + u(5), 48 + u(10))	0	`name_len`
	u(8)	0	`name[name_len]`
`!d_alpha and type ==` **kAlpha**	Bool()	`false`	`alpha_associated`

Table D.8 *(continued)*

condition	type	default	name
`type` == **kSpotColour**	F16()	0	`red`
`type` == **kSpotColour**	F16()	0	`green`
`type` == **kSpotColour**	F16()	0	`blue`
`type` == **kSpotColour**	F16()	0	`solidity`
`type` == **kCFA**	U32(1, u(2), 3 + u(4), 19 + u(8))	1	`cfa_channel`

Table D.9 specifies the meaning of the ExtraChannelType values.

Table D.9 — ExtraChannelType

name	value	meaning
kAlpha	0	Alpha transparency, where 0 means fully transparent
kDepth	1	Depth map. Higher values mean farther away from the camera.
kSpotColour	2	Spot colour channel; red, green, blue indicate its colour and solidity in [0, 1] indicates the overall blending factor, with 0 corresponding to fully translucent (invisible) and 1 corresponding to fully opaque.
kSelectionMask	3	Selection mask, which indicates a (fuzzy) region of interest, for example for image manipulation purposes. Pixels with value zero do not belong to the selection, pixels with the maximum value do belong to the selection.
kBlack	4	The K channel of a CMYK image. If present, a CMYK ICC profile is also present, and the RGB samples are to be interpreted as CMY, where 0 denotes full ink.
kCFA	5	Channel used to represent Colour Filter Array data (Bayer mosaic)
kThermal	6	Infrared thermography image. Higher values mean warmer temperature.
kNonOptional	15	The decoder indicates it cannot safely interpret the semantics of this extra channel.
kOptional	16	Extra channel that can be safely ignored.

Extra channels are interpreted according to their type and are rendered as specified in L.4.

`dim_shift` is the base-2 logarithm of the downsampling factor of the dimensions of the extra channel with respect to the main image dimensions defined in size. The dimensions of the extra channel are rounded up. The value 1 << `dim_shift` does not exceed the `group_dim` of any frame (F.2).

EXAMPLE `dim_shift == 3` implies 8×8 downsampling, e.g., a 3 × 3 extra channel for a 20 × 20 main image.

If `name_len` > 0, then the extra channel has a name `name`, interpreted as a UTF-8 encoded string.

Annex E
(normative)

Colour encoding

E.1 General

The interpretation of RGB pixel data is governed by a colour space described by a colour profile, for example sRGB is a colour space. Capture devices, display devices, and quantized (integer) pixel data can all have their own colour space. Converting quantized pixels in one colour space to quantized pixels in a different colour space is a lossy operation.

Pixel data can be encoded in two ways: either in the absolute XYB colour space, or as RGB triplets, YCbCr triplets or greyscale values which are to be interpreted according to the colour space described by a given colour profile.

NOTE Both ways are useful in different scenarios:

— XYB is designed to match the human visual system and is excellent for lossy compression.

— When performing lossless compression, integer data has to be kept in its original colour space, since converting it and re-quantizing it in another colour space is lossy.

A colour space is always signalled in the metadata, and it has a different meaning in each scenario:

— In the XYB scenario, the signalled colour space does not give any information about how to interpret the encoded XYB pixels, since they are already in an absolute colour space. It is merely intended to indicate what colour space the original pixel data had, which can be useful when converting the decoded image to integer data. To display the image data on a display device with a display profile, this is irrelevant since it suffices to convert from XYB to the display colour space. The signalled colour space can be unused in that case. Only in case there are multiple frames that need to be blended by the decoder, the signalled colour space is relevant (since the blending may have to be done in that colour space and not in XYB, see F.2).

— In the other scenario, the signalled colour space has direct meaning: it is the colour space of the RGB, YCbCr or greyscale pixel data. To display the image on a display device, the pixel data is to be converted from the signalled colour space to the display colour space.

The `metadata.xyb_encoded` flag (D.3) indicates which of the two scenarios is followed. In the case where `metadata.xyb_encoded` is `true`, then `metadata.colour_encoding`, `metadata.bit_depth`, and the ICC profile described in this Annex (if present) are merely suggestions as to a colour encoding and sample representation to use after the intermediate conversion to the linear sRGB colour space specified in L.2 (unless the frame is saved as a reference frame and `save_before_ct` is `false`, in which case it is not merely a suggestion but the colour space in which blending is performed, see F.2). If `xyb_encoded` is `false`, the decoded samples are to be interpreted according to `metadata.bit_depth` and the colour space defined in the following way:

— If `!metadata.colour_encoding.want_icc`, the decoded samples are indicated to be interpreted according to the `colour_space`, `white_point`, `primaries`, `transfer_function` and `rendering_intent` of `metadata.colour_encoding`, as specified in E.2.

— Otherwise, the decoded samples are indicated to be interpreted according to an ICC profile decoded as specified in E.4.

EXAMPLE In this example, an original image is in the DCI-P3 colour space. In the case of lossless compression, an encoder can signal the DCI-P3 colour space using `metadata.colour_encoding`, set `metadata.xyb_encoded` to `false`, and encode the pixel values as they are. For lossy compression, an encoder can also signal the DCI-P3 colour space using `metadata.colour_encoding`, set `metadata.xyb_encoded` to `true`, and convert the pixel values to the XYB colour space before encoding them. In this case, a decoder decodes the image in the XYB colour space, and applies L.2 to convert it to linear sRGB, which is an intermediate representation where some of the sample values can be negative or higher than the nominal maximum value since they are outside the sRGB gamut. To display the image on an sRGB display device, it can convert this intermediate representation to the sRGB colour space (i.e. apply clamping and adjust the transfer curve). To save the image in another (integer-based) image format, it can convert the intermediate representation to the suggested DCI-P3 colour space that was signalled, quantize the sample values to integers at the bit depth that was signalled, and save the resulting image.

E.2 ColourEncoding

The ColourEncoding bundle is specified in Table E.1. It references Table E.2 (Customxy), Table E.3 (ColourSpace), Table E.4 (WhitePoint), Table E.5 (Primaries), Table E.6 (TransferFunction), Table E.7 (CustomTransferFunction), and Table E.8 (RenderingIntent).

Table E.1 — ColourEncoding bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	Bool()	`false`	`want_icc`
`!all_default`	Enum(ColourSpace)	**kRGB**	`colour_space`
`use_desc and not_xyb`	Enum(WhitePoint)	**kD65**	`white_point`
`white_point == WP.kCustom`	Customxy		`white`
`has_primaries`	Enum(Primaries)	PR.**kSRGB**	`primaries`
`primaries == PR.kCustom`	Customxy		`red`
`primaries == PR.kCustom`	Customxy		`green`
`primaries == PR.kCustom`	Customxy		`blue`
`use_desc`	CustomTransferFunction		`tf`
`use_desc`	Enum(RenderingIntent)	**kRelative**	`rendering_intent`

In Table E.1, CS denotes ColourSpace, WP denotes WhitePoint, PR denotes Primaries, `use_desc` expands to `!all_default and !want_icc`, `not_xyb` expands to `colour_space !=` **`kXYB`**, and `has_primaries` expands to `use_desc and not_xyb and colour_space !=` **`kGrey`**.

`want_icc` is `true` if and only if the codestream stores an ICC profile. If so, it is decoded as specified in E.4, and describes the colour space. Also, `colour_space` is **kGrey** if and only if the ICC profile is greyscale. Otherwise, there is no ICC profile and the other fields describe the colour space.

Table E.2 — Customxy bundle

condition	type	name
	U32(u(19), 524288 + u(19), 1048576 + u(20), 2097152 + u(21))	`ux`
	U32(u(19), 524288 + u(19), 1048576 + u(20), 2097152 + u(21))	`uy`

`x = UnpackSigned(ux)` and `y = UnpackSigned(uy)` are the coordinates of a point on the CIE xy chromaticity diagram, scaled by 10^6. The unscaled coordinates may be outside [0, 1] for imaginary primaries.

Table E.3 — ColourSpace

name	value	meaning
kRGB	0	Tristimulus RGB, with given white point and primaries. This includes CMYK colour spaces; in that case, the RGB components are interpreted as CMY where 0 means full ink, `want_icc` is `true` (see Table E.1), and there is an extra channel of type **kBlack** (see Table D.9).
kGrey	1	Luminance, with a given white point; `colour_encoding.primaries` are not read in this case
kXYB	2	XYB (opsin); `colour_encoding.white_point` is **kD65**, `colour_encoding.primaries` is not read, `colour_encoding.have_gamma` is `true`, `colour_encoding.gamma` is 3333333
kUnknown	3	None of the other table entries describe the colour space appropriately

Table E.4 — WhitePoint

name	value	meaning
kD65	1	CIE Standard Illuminant D65: 0.3127, 0.3290
kCustom	2	Custom white point stored in `colour_encoding.white`
kE	10	CIE Standard Illuminant E (equal-energy): 1/3, 1/3
kDCI	11	DCI-P3 from SMPTE ST 428-1: 0.314, 0.351

The meaning column is interpreted as CIE xy chromaticity coordinates.

NOTE 1 The values are a selection of the values defined in ISO/IEC 23091-2.

Table E.5 — Primaries

name	value	meaning
kSRGB	1	0.639998686, 0.330010138; 0.300003784, 0.600003357; 0.150002046, 0.059997204
kCustom	2	Custom red/green/blue primaries, which are stored in `colour_encoding.red/green/blue`
k2100	9	0.708, 0.292; 0.170, 0.797; 0.131, 0.046 (As specified in ITU-R BT.2100-2)
kP3	11	0.680, 0.320; 0.265, 0.690; 0.150, 0.060 (As specified in SMPTE ST 428-1)

The values in the meaning column are interpreted as CIE xy chromaticity coordinates: red; green; blue, respectively.

NOTE 2 The values are a selection of the values defined in ISO/IEC 23091-2. The `xy` coordinates of **kSRGB** are quantized and match the values that would be stored in an ICC profile.

Table E.6 — TransferFunction

name	value	meaning
k709	1	As specified in ITU-R BT.709-6
kUnknown	2	None of the other table entries describe the transfer function
kLinear	8	The gamma exponent is 1
kSRGB	13	As specified in IEC 61966-2-1
kPQ	16	As specified in ITU-R BT.2100-2 (PQ)
kDCI	17	As specified in SMPTE ST 428-1
kHLG	18	As specified in ITU-R BT.2100-2 (HLG)

NOTE 3 The values are a selection of the values defined in ISO/IEC 23091-2.

NOTE 4 Non-HLG signals are interpreted as being display-referred, and their conversion from and to HLG should therefore include the HLG OOTF and inverse OOTF respectively, using `intensity_target` as the peak display luminance L_W and computing γ as `1.2 * pow(1.111, log2(L_W / 1000))`.

Table E.7 — CustomTransferFunction bundle

condition	type	default	name
	Bool()	`false`	`have_gamma`
`have_gamma`	u(24)		`gamma`
`!have_gamma`	Enum(TransferFunction)	**kSRGB**	`transfer_function`

`have_gamma == false` indicates the transfer function is defined by `transfer_function`. Otherwise, the opto-electrical transfer function is characterized by the exponent `gamma` / 10^7. This exponent is in (0, 1].

Table E.8 — RenderingIntent

name	value	meaning
kPerceptual	0	As specified in ISO 15076-1 (vendor-specific)
kRelative	1	As specified in ISO 15076-1 (media-relative)
kSaturation	2	As specified in ISO 15076-1 (vendor-specific)
kAbsolute	3	As specified in ISO 15076-1 (ICC-absolute)

NOTE 5 The values are defined by ISO 15076-1.

E.3 ToneMapping

The ToneMapping bundle is specified in Table E.9.

Table E.9 — ToneMapping bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	F16()	255	`intensity_target`
`!all_default`	F16()	0	`min_nits`
`!all_default`	Bool()	`false`	`relative_to_max_display`
`!all_default`	F16()	0	`linear_below`

`intensity_target` is > 0. This is an upper bound on the intensity level present in the image in nits, and represents the intensity corresponding to the value "1.0" (but the image need not contain a pixel this bright).

`min_nits` is `>= 0` and `<= intensity_target`. This is a lower bound on the intensity level present in the image in nits, but the image need not contain a pixel this dark.

`linear_below` represents a value below which tone mapping leaves the values unchanged. If `relative_to_max_display` is `true`, `linear_below` is a ratio in [0, 1] of the maximum display brightness in nits. Otherwise, it is an absolute brightness in nits `>= 0`.

E.4 ICC profile

This clause specifies how to decode an ICC profile.

NOTE These are standard display ICC profiles as defined by the International Colour Consortium.

E.4.1 Data stream

The decoder reads `enc_size` as U64(). The decoder reads 41 pre-clustered distributions as specified in C.1. The decoder then reads `enc_size` integers as specified in C.3.3 to obtain decompressed bytes in the range [0,255], using `DecodeHybridVarLenUint(IccContext(index, prev_byte, prev_prev_byte))` where `index` is the current byte index, `prev_byte` and `prev_prev_byte` are respectively the previous and second-previous bytes or 0 if they do not exist yet, and the `IccContext()` function is defined in the code below. The resulting decompressed byte-based data is the encoded ICC stream. The decoder reconstructs the ICC profile from this encoded ICC stream as described in the subsequent subclauses of this annex.

```
IccContext(i, b1, b2) {
  if (i <= 128) return 0;
  if (b1 >= 'a' and b1 <= 'z') p1 = 0;
  else if (b1 >= 'A' and b1 <= 'Z') p1 = 0;
  else if (b1 >= '0' and b1 <= '9') p1 = 1;
  else if (b1 == '.' or b1 == ',') p1 = 1;
  else if (b1 <= 1) p1 = 2 + b1;
  else if (b1 > 1 and b1 < 16) p1 = 4;
  else if (b1 > 240 and b1 < 255) p1 = 5;
  else if (b1 == 255) p1 = 6;
  else p1 = 7;
  if (b2 >= 'a' and b2 <= 'z') p2 = 0;
  else if (b2 >= 'A' and b2 <= 'Z') p2 = 0;
  else if (b2 >= '0' and b2 <= '9') p2 = 1;
  else if (b2 == '.' or b2 == ',') p2 = 1;
  else if (b2 < 16) p2 = 2;
  else if (b2 > 240) p2 = 3;
  else p2 = 4;
  return 1 + p1 + p2 * 8;
}
```

E.4.2 Encoded ICC stream

In this subclause and the following ones, Varint() reads an unsigned integer value of up to 63 bits as specified by the following code:

```
value = 0; shift = 0;
while (1) {
  /* shift <= 56 */
  b = u(8);
  value += (b & 127) << shift;
  if (b <= 127) break;
  shift += 7;
}
```

The encoded ICC stream is structured as shown in Table E.10. In this table, Varint() and u() are read from the encoded ICC stream, not from the JPEG XL codestream.

Table E.10 — ICC stream

type	name	comment
Varint()	`output_size`	the size of the decoded ICC profile, in bytes
Varint()	`commands_size`	the size of the command stream, in bytes
u(8 * commands_size)	command stream	sub-stream of the encoded ICC stream which contains command bytes
u(8 * [remaining bytes])	data stream	sub-stream of the encoded ICC stream which contains data bytes, starting directly after the command stream and ending at the end of the encoded ICC stream

The command stream is structured as shown in Table E.11. In this table, u() is read from the command stream in the encoded ICC stream, not from the JPEG XL codestream.

Table E.11 — command stream

type	comment
u(8 * [variable amount])	commands for decoding tag list
u(8 * [remaining bytes])	commands for decoding main content

The data stream is structured as shown in Table E.12. In this table, u() is read from the encoded data stream in the encoded ICC stream, not from the JPEG XL codestream.

Table E.12 — data stream

type	comment
u(8 * [variable amount])	data for decoding ICC header
u(8 * [variable amount])	data for decoding ICC tag list
u(8 * [remaining bytes])	data for decoding maincontent

The resulting ICC profile, output by the decoder, is structured as shown in Table E.13. In this table, u() is read from the resulting ICC profile, not the JPEG XL codestream.

Table E.13 — ICC profile

type	name	comment
u(8 * [variable amount])	ICC header	up to 128 bytes long. Decoding procedure specified in E.4.3.
u(8 * [variable amount])	ICC tag list	Decoding procedure specified in E.4.4.
u(8 * [remaining bytes])	main content	Decoding procedure specified in E.4.5.

After decoding the `output_size` and `commands_size`, the encoded ICC stream is considered to be divided into two input streams: the command stream, which starts at the current position and runs for `commands_size` bytes, and the data stream, which starts directly after the command stream and contains all remaining bytes. The decoder maintains the current positions within the command stream and the data stream, initially at the beginning of each respective stream, and increment each when a byte from the corresponding stream is read as indicated in the next subclauses. A stream position does not go beyond the last byte of that stream, and the commands stream does not extend beyond the end of the encoded ICC stream.

As described in the next subclauses, the decoder decodes an ICC header (E.4.3), ICC tag list (E.4.4), and main content (E.4.5) from those two streams. The decoded ICC profile is the concatenation of these three parts in that order, which is indicated in the next subclauses as appending bytes to the result or output. The ICC tag list and main content can be empty if the decoder finishes earlier.

If at any time in E.4.3, E.4.4 or E.4.5, the ICC decoder is finished, then it returns the result as the ICC profile. The decoder reads all bytes from the data stream as well as all bytes from the command stream (that is, it reads exactly `commands_size` bytes from the command stream and reads exactly all remaining bytes from the data stream). The size of the resulting ICC profile is exactly `output_size` bytes.

E.4.3 ICC header

The header is the first of three concatenated parts that the decoder outputs.

The `header_size` in bytes is `min(128, output_size)`. The decoder reads `header_size` bytes from the data stream. For each byte `e`, it computes a prediction `p` as indicated below, then computes an output header byte as `(p + e) & 255` and appends it to the ICC result.

Each predicted value `p` is computed as specified by the following code, where `i` is the position in the header of the current byte, starting from 0. `header[j]` refers to the earlier output header byte at position `j`. Single characters, and characters in strings, point to ASCII values.

```
if (i == 0 or i == 1 or i == 2 or i == 3)
  // 'output_size[i]' means byte i of output_size encoded as an
  // unsigned 32-bit integer in big endian order
  p = output_size[i];
else if (i == 8) p = 4;
else if (i >= 12 and i <= 23) {
  s = "mntrRGB XYZ ";  // one space after "RGB" and one after "XYZ"
  p = s[i - 12]; }
else if (i >= 36 and i <= 39) { s = "acsp"; p = s[i - 36]; }
else if ((i == 41 or i == 42) and header[40] == 'A') p = 'P';
else if (i == 43 and header[40] == 'A') p = 'L';
else if (i == 41 and header[40] == 'M') p = 'S';
else if (i == 42 and header[40] == 'M') p = 'F';
else if (i == 43 and header[40] == 'M') p = 'T';
else if (i == 42 and header[40] == 'S' and header[41] == 'G') p = 'I';
else if (i == 43 and header[40] == 'S' and header[41] == 'G') p = 32;
else if (i == 42 and header[40] == 'S' and header[41] == 'U') p = 'N';
else if (i == 43 and header[40] == 'S' and header[41] == 'U') p = 'W';
else if (i == 70) p = 246;
else if (i == 71) p = 214;
else if (i == 73) p = 1;
else if (i == 78) p = 211;
else if (i == 79) p = 45;
else if (i >= 80 and i < 84) p = header[4 + i - 80];
else p = 0;
```

If `output_size` is smaller than or equal to 128, then the above procedure has produced the full output, the ICC decoder is finished and the remaining subclauses are skipped.

E.4.4 ICC tag list

The ICC tag list is the second of three concatenated output parts that the decoder outputs to the resulting ICC profile. The decoder keeps reading from the same command stream and data stream as before, continuing at the positions reached at the end of the previous subclause.

To decode the tag list, the decoder reads the number of tags as specified by the following code. If the end of the command stream is reached, the decoder is finished, the full ICC profile is decoded, the decoder ends this procedure and skips the next subclause.

```
v = Varint(); /* from command stream */
num_tags = v - 1;
if (num_tags == -1) {
 /* output nothing, stop reading the tag list, and proceed to E.4.5 */
}
/* Append num_tags to the output as a big endian unsigned 32-bit integer (4 bytes) */
previous_tagstart = num_tags * 12 + 128;
previous_tagsize = 0;
```

Then, the decoder repeatedly reads a tag as specified by the following code until a tag with tagcode equal to 0 is read or until the end of the command stream is reached.

```
command = u(8); /* from command stream */
tagcode = command & 63;
if (tagcode == 0) {
 /* decoding the tag list is done, proceed to E.4.5 */
}
tag = ""; // 4-byte string with tag name
if (tagcode == 1) {
  // the tag string is set to 4 custom bytes read from the data
  tag = u(4 * 8) from data stream;
} else if (tagcode == 2) {
  tag = "rTRC";
} else if (tagcode == 3) {
  tag = "rXYZ";
} else if (tagcode >= 4 and tagcode < 21) {
  // the tag string is set to one of the predefined values
  strings = {
    "cprt", "wtpt", "bkpt", "rXYZ", "gXYZ", "bXYZ", "kXYZ", "rTRC", "gTRC",
    "bTRC", "kTRC", "chad", "desc", "chrm", "dmnd", "dmdd", "lumi"
  };
  tag = strings[tagcode - 4];
} else {
  /* this branch is not reached */
}
tagstart = previous_tagstart + previous_tagsize;
if ((command & 64) != 0) tagstart = Varint(); /* from command stream */
tagsize = previous_tagsize;
if (tag == "rXYZ" or tag == "gXYZ" or tag == "bXYZ" or tag == "kXYZ" or
    tag == "wtpt" or tag == "bkpt" or tag == "lumi") {
  tagsize = 20;
}
if ((command & 128) != 0) tagsize = Varint(); /* from command stream */
previous_tagstart = tagstart;
```

```
previous_tagsize = tagsize;
```

The procedure appends the computed ICC tag(s) to the output as specified by the following code. Any integer values such as tagsize, tagstart and any operations on them are always appended as big endian 32-bit integers (4 bytes), when appended to the output.

```
/* append the following 3 groups of 4 bytes to the output: tag, tagstart, tagsize */
if (tagcode == 2) {
  /* append the following additional 6 groups of 4 bytes to the output:
    "gTRC", tagstart, tagsize, "bTRC", tagstart, tagsize */
} else if (tagcode == 3) {
  /* append the following additional 6 groups of 4 bytes to the output:
    "gXYZ", (tagstart + tagsize), tagsize, "bXYZ", (tagstart + 2 * tagsize), tagsize */
}
```

E.4.5 Main content

The main content is the third of three concatenated output parts that the decoder outputs to the resulting ICC profile. The decoder keeps reading from the same command stream and data stream as before, continuing at the positions reached at the end of the previous subclause.

In this subclause, `Shuffle(bytes, width)` denotes the following operation: Bytes are inserted in raster order (row by row from left to right, starting with the top row) into a matrix with `width` rows, and as many columns as needed. The last column can have missing elements at the bottom if len(`bytes`) is not a multiple of `width`. Those elements are skipped and no byte is taken from the input for these missing elements: the input is the concatenation of all rows excluding the possibly missing last element of each row. Then the matrix is transposed, and bytes are overwritten with elements of the transposed matrix read in raster order, not including the missing elements which are now at the end of the last row.

EXAMPLE Shuffling the list (1, 2, 3, 4, 5, 6, 7) with `width` 2 results in the list (1, 5, 2, 6, 3, 7, 4).

To decode the main content, the decoder reads from the command stream as specified by the following code, until the end of the command stream is reached (which can be immediately).

```
command = u(8); /* from command stream */
if (command == 1) {
  num = Varint(); /* from command stream; num > 0 */
  bytes = u(num * 8); /* from data stream */
  /* append bytes to output_stream */
} else if (command == 2 or command == 3) {
  num = Varint(); /* from command stream; num > 0 */
  bytes = u(num * 8); /* from data stream */
  width = (command == 2) ? 2 : 4;
  Shuffle(bytes, width);
  /* append bytes to output_stream */
} else if (command == 4) {
  flags = u(8); /* from command stream */
  width = (flags & 3) + 1;     /*  width != 3  */
  order = (flags & 12) >> 2;   /*  order != 3  */
  stride = width;
  if ((flags & 16) != 0) { stride = Varint(); /* from command stream */ }
  /* stride * 4 < number of bytes already output to ICC profile */
  /* stride >= width  */
  num = Varint(); /* from command stream; num > 0 */
  bytes = u(num * 8); /* from data stream */
  if (width == 2 or width == 4) {
    Shuffle(bytes, width);
  }
  // Run an Nth-order predictor on num bytes as follows, with the bytes
  // representing unsigned integers of the given width (but num is not
  // required to be a multiple of width).
  for (i = 0; i < num; i += width) {
    N = order + 1;
    prev = /* N-element array of unsigned integers of width bytes each */
    for (j = 0; j < N; ++j) {
      prev[j] = /* read u(width * 8) from the output ICC profile starting from
                   (stride * (j + 1)) bytes before the current output size,
                   interpreted as a big-endian unsigned integer of width bytes */
    }
    // Compute predicted number p, where p and elements of prev are
    // unsigned integers of width bytes each.
    if (order == 0) p = prev[0];
    else if (order == 1) p = 2 * prev[0] - prev[1];
    else if (order == 2) p = 3 * prev[0] - 3 * prev[1] + prev[2];
    for (j = 0; j < width and i + j < num; ++j) {
      val = (bytes[i + j] + (p >> (8 * (width - 1 - j)))) & 255;
      /* append val as 1 byte to the ICC profile output */
    }
  }
} else if (command == 10) {
  /* append "XYZ " (4 ASCII characters) to the output */
  /* append 4 bytes with value 0 to the output */
  bytes = u(12 * 8); /* from data stream */
  /* append bytes to the output */
} else if (command >= 16 and command < 24) {
  strings = {"XYZ ", "desc", "text", "mluc", "para", "curv", "sf32", "gbd "};
  /* append 4 bytes from strings[command - 16] to the output */
  /* append 4 bytes with value 0 to the output */
} else {
  /* this branch is not reached */
}
```

Annex F
(normative)

Frame header

F.1 General

Subsequent to headers (Annex D) and ICC profile, if present (E.4), the codestream consists of one or more frames. Each Frame is byte-aligned by invoking ZeroPadToByte() (B.2.7). It is read according to Table F.1.

NOTE The LF coefficients (G.2.2) always correspond to a 1:8 downsampled image, regardless of the sizes of the var-DCT blocks that were used. In the case of DCT8x8, these are the DC coefficients. In the case of bigger block sizes, these also include information derived from the low-frequency AC coefficients (which are recovered using the procedures described in I.8), and these low-frequency AC coefficients are skipped in the HF encoding (I.4). In the case of smaller block sizes like DCT4x4, the “LF coefficient” is effectively the average of the smaller-block DC coefficients, and the HF encoding effectively contains the remaining information to restore the smaller-block DC.

Table F.1 — Frame bundle

condition	type	name	subclause
	FrameHeader	`frame_header`	F.2
	TOC	`toc`	F.3
	LfGlobal	`lf_global`	G.1
	LfGroup	`lf_group[num_lf_groups]`	G.2
`encoding ==` **kVarDCT**	HfGlobal	`hf_global`	G.3
	PassGroup	`group_pass[num_groups * num_passes]`	G.4

The order of the sections of Table F.1 that come after TOC, is the conceptual order after undoing the optional permutation (see F.3). The actual bitstream order can be different due to this permutation.

The dimensions of a frame in pixels (`width` and `height`) are `size.width` and `size.height` if `!frame_header.have_crop`, otherwise they are given by `frame_header.width` and `frame_header.height`. These are interpreted according to the sample grid before taking `metadata.orientation` into account. If `upsampling > 1` (which is a field in `frame_header`), then `width = ceil(width / upsampling)` and `height = ceil(height / upsampling)`. If `lf_level > 0` (which is also a field in `frame_header`), then `width = ceil(width / (1 << (3 * lf_level)))`. and `height = ceil(height / (1 << (3 * lf_level)))`.

Let `num_groups` be equal to `ceil(width/group_dim) * ceil(height/group_dim)`, where `group_dim` is the width and height of a group (5.3). `num_lf_groups` denotes `ceil(width / (group_dim * 8)) * ceil(height / (group_dim * 8))`. Note that `encoding`, `group_dim` and `num_passes` are fields in `frame_header`.

F.2 FrameHeader

The decoder reads FrameHeader as specified in Table F.2.

Table F.2 — FrameHeader bundle

condition	type	default	name
	Bool()	true	all_default
!all_default	u(2)	**kRegularFrame**	frame_type
!all_default	u(1)	**kVarDCT**	encoding
!all_default	U64()	0	flags
!all_default and !metadata.xyb_encoded	Bool()	false	do_YCbCr
do_YCbCr and !flags.**kUseLfFrame**	u(2)	0	jpeg_upsampling[3]
!all_default and !flags.**kUseLfFrame**	U32(1, 2, 4, 8)	1	upsampling
!all_default and !flags.**kUseLfFrame**	U32(1, 2, 4, 8)	1	ec_upsampling[num_extra]
encoding == **kModular**	u(2)	1	group_size_shift
!all_default and metadata.xyb_encoded and encoding == **kVarDCT**	u(3)	d_xqms	x_qm_scale
!all_default and metadata.xyb_encoded and encoding == **kVarDCT**	u(3)	2	b_qm_scale
!all_default and frame_type != **kReferenceOnly**	Passes		passes
frame_type == **kLFFrame**	1 + u(2)	0	lf_level
!all_default and frame_type != **kLFFrame**	Bool()	false	have_crop
have_crop and frame_type != **kReferenceOnly**	U32(u(8), 256 + u(11), 2304 + u(14), 18688 + u(30))	0	ux0
have_crop and frame_type != **kReferenceOnly**	U32(u(8), 256 + u(11), 2304 + u(14), 18688 + u(30))	0	uy0
have_crop	U32(u(8), 256 + u(11), 2304 + u(14), 18688 + u(30))	0	width
have_crop	U32(u(8), 256 + u(11), 2304 + u(14), 18688 + u(30))	0	height
!all_default and normal_frame	BlendingInfo		blending_info
!all_default and normal_frame	BlendingInfo		ec_blending_info[num_extra]
!all_default and normal_frame and metadata.have_animation	U32(0, 1, u(8), u(32))	0	duration
!all_default and normal_frame and metadata.animation.have_timecodes	u(32)	0	timecode
!all_default and normal_frame	Bool()	!frame_type	is_last
!all_default and frame_type != **kLFFrame** and !is_last	u(2)	0	save_as_reference
!all_default and (frame_type == **kReferenceOnly** or (resets_canvas and can_reference))	Bool()	!normal_frame	save_before_ct
!all_default	U32(0, u(4), 16 + u(5), 48 + u(10))	0	name_len

Table F.2 *(continued)*

condition	type	default	name
	u(8)	0	`name[name_len]`
`!all_default`	RestorationFilter (J.1)		`restoration_filter`
`!all_default`	Extensions		`extensions`

Let `normal_frame` denote the expression `(frame_type == `**kRegularFrame**` or frame_type == `**kSkipProgressive**`)`. Let `full_frame` be `true` if and only if `have_crop` is `false` or if the frame area given by `width` and `height` and offsets `x0` and `y0` completely covers the image area. Let `resets_canvas` denote the expression `full_frame and blending_info.mode == `**kReplace**. Let `can_reference` denote the expression `!is_last and (duration == 0 or save_as_reference != 0) and frame_type != `**kLFFrame**.

`frame_type` is defined in Table F.3.

Table F.3 — FrameHeader.frame_type

name	value	meaning
kRegularFrame	0	A regular frame, which is part of the decoded sequence of frames.
kLFFrame	1	Represents the LF of a future frame. Is not itself part of the decoded sequence of frames.
kReferenceOnly	2	Frame will only be used as a source for Patches or frame blending. Is not itself part of the decoded sequence of frames.
kSkipProgressive	3	Same as **kRegularFrame** in terms of the (final) decoded sequence of frames, but decoders do not progressively render previews of frames of this type.

`encoding` is defined in Table F.4.

Table F.4 — FrameHeader.encoding

name	value	meaning
kVarDCT	0	Var-DCT mode: the decoder will perform IDCT on varblocks to obtain the decoded image.
kModular	1	Modular mode: the decoder will perform a signalled chain of zero or more inverse (typically reversible) transforms.
NOTE Var-DCT mode is suitable for lossy encodings and for lossless JPEG recompression; Modular mode is suitable for lossless encodings.		

Both encodings split each frame into groups. This document specifies the meaning of the encodings (e.g., **kVarDCT**) where they are referenced.

`flags` is defined in Table F.5.

Table F.5 — FrameHeader.flags

name	value	meaning
kNoise	xxxx xxx1	Enable noise feature stage
kNoise	xxxx xxx0	Disable noise feature stage
kPatches	xxxx xx1x	Enable patch feature stage
kPatches	xxxx xx0x	Disable patch feature stage
kSplines	xxx1 xxxx	Enable spline feature stage
kSplines	xxx0 xxxx	Disable spline feature stage
kUseLfFrame	xx1x xxxx	Enable use of special LF frames
kUseLfFrame	xx0x xxxx	Disable use of special LF frames
kSkipAdaptiveLFSmoothing	0xxx xxxx	Enable adaptive LF smoothing
kSkipAdaptiveLFSmoothing	1xxx xxxx	Disable adaptive LF smoothing

The bits of `frame_header.flags` encode whether certain features are enabled or disabled in the current frame, as specified in references to this subclause.

EXAMPLE 1 If `flags == 18`, the decoder enables spline rendering and patch rendering.

When a flag name is used, its value is understood as the value of the corresponding bit (0 or 1).

EXAMPLE 2 If `frame_header.flags == 3`, **kPatches** is considered to be 1.

If `do_YCbCr`, then the pixels are stored in YCbCr, and are converted to RGB as specified in L.3.

NOTE 1 In addition, one or more reversible colour transforms can be applied in Modular mode (see H.6.3, H.6.4). These transforms are not signalled in the `FrameHeader`, but in the `ModularHeader` (H.2), and can be applied either to the whole frame or on a per-group basis. These are always undone, regardless of the value of `save_before_ct`.

`jpeg_upsampling` indicates the degree of subsampling for each YCbCr channel. 0 denotes {horizontal, vertical} subsampling factors of {1, 1} for the given channel; 1 denotes {2, 2}, 2 denotes {2, 1} and 3 denotes {1, 2}. The horizontal and vertical size, in 8×8 blocks, of each channel is divided (rounding up) by the maximum subsampling factor across all channels, and then multiplied by the subsampling factor for the current channel.

`upsampling` and `ec_upsampling` also indicate subsampling, by a factor of 1, 2, 4 or 8 (both horizontally and vertically), for respectively the colour image and the extra channels. In the case of extra channels this subsampling is cumulative with the subsampling implied by `dim_shift`; for all extra channels, the cumulative upsampling factor `ec_upsampling[i] << ec_info[i].dim_shift` shall not exceed 64.

If `upsampling > 1`, then for all extra channels, `(ec_upsampling[i] << ec_info[i].dim_shift) >= upsampling`.

The division in groups is not affected by `jpeg_upsampling` and `ec_upsampling`: even if a channel is subsampled, group splitting proceeds according to the location of samples and blocks in the image, after subsampling by a factor of upsampling. For example, if `jpeg_upsampling` is equal to {0, 1, 1}, i.e. 4:2:0 chroma subsampling, and there is an alpha channel for which `ec_upsampling` is 4, then the groups consist of 256×256 Y samples, 128×128 Cb and Cr samples, and 64×64 alpha samples, which all correspond to the same image region.

After decoding subsampled channels, they are upsampled as specified in J.2 (in the case of `jpeg_upsampling`) and K.2 (in the case of `upsampling` and `ec_upsampling`).

The value of `group_dim` is set to 128 << `group_size_shift`.

The default value `d_xqms` for `x_qm_scale` is defined as follows: `d_xqms = (metadata.xyb_encoded and encoding ==` **`kVarDCT`** `? 3 : 2)`.

Passes are defined in Table F.6.

Table F.6 — Passes bundle

condition	type	default	name
	U32(1, 2, 3, 4 + u(3))	1	`num_passes`
`num_passes != 1`	U32(0, 1, 2, 3 + u(1))	0	`num_ds`
`num_passes != 1`	u(2)	0	`shift[num_passes-1]`
`num_passes != 1`	U32(1, 2, 4, 8)	1	`downsample[num_ds]`
`num_passes != 1`	U32(0, 1, 2, u(3))	0	`last_pass[num_ds]`

`num_passes` indicates the number of passes into which the frame is partitioned.

`num_ds` indicates the number of (`downsample`, `last_pass`) pairs between 0 and 4, inclusive. It is strictly smaller than `num_passes`.

`shift[i]` indicates the amount by which the HF coefficients of the pass with index i in the range [0, `num_passes`) are left-shifted immediately after entropy decoding. The last pass behaves as if it had a shift value of 0.

With `i` in the range [0, `num_ds`), `downsample[i]` indicates a downsampling factor in both x and y directions, and `last_pass[i]` indicates the zero-based index of the final pass that is to be decoded when the intention is to display the image in a downsampled way with a downsampling factor `downsample[i]`. This index is at most `num_passes` − 1. The sequence `downsample` is strictly decreasing and the sequence `last_pass` is strictly increasing.

In addition to the (`downsample`, `last_pass`) pairs that are explicitly encoded in the codestream, the decoder behaves as if a final pair equal to (1, `num_passes` − 1) were present.

If `lf_level != 0`, the samples of the frame (before any colour transform is applied) are recorded as LFFrame[`lf_level`−1] and may be referenced by subsequent frames.

If `can_reference`, then the samples of the decoded frame are recorded as Reference[`save_as_reference`] and may be referenced by subsequent frames. The decoded samples are recorded before any colour transform (XYB or YCbCr) if `save_before_ct` is `true`, and after colour transform otherwise (in which case they are converted to the RGB colour space signalled in the image header). Blending is performed before recording the reference frame. If `!save_before_ct and frame_type ==` **kReferenceOnly**, then `width == size.width` and `height == size.height`.

If `have_crop`, the decoder considers the current frame to have dimensions `width` × `height`, and updates the rectangle of the image with top-left corner `x0 = UnpackSigned(ux0)`, `y0 = UnpackSigned(uy0)` with the current frame using the given `blend_mode`. If `x0` or `y0` is negative, or the frame extends beyond the right or bottom edge of the image, only the intersection of the frame with the image is updated and contributes to the decoded image. If `!have_crop`, the frame has the same dimensions as the image.

Let `extra` be `true` if and only if and the number of extra channels is at least one. `blendinginfo` and `ec_blendinginfo` are defined in Table F.7.

Table F.7 — BlendingInfo bundle

condition	type	default	name
	U32(0, 1, 2, 3 + u(2))	0	`mode`
`extra and (mode ==` **kBlend** `or mode ==` **kMulAdd** `)`	U32(0, 1, 2, 3 + u(3))	0	`alpha_channel`
`extra and (mode ==` **kBlend** `or mode ==` **kMulAdd** `or mode ==` **kMul** `)`	Bool()	`false`	`clamp`
`!resets_canvas`	u(2)	0	`source`

The mode values are defined in Table F.8. All blending operations consider as “previous sample” the sample at the corresponding coordinates in the source frame, which is the frame that was previously stored in Reference[`source`] – if no frame was previously stored, the source frame is assumed to have all sample values set to zeroes. The `blend_info` affects the three colour channels. The extra channels are blended according to `ec_blend_info` instead. The blending is done in the colour space *after* inverse colour transforms from Annex L have been applied (except for L.4). For blend modes **kBlend** and **kMulAdd**, “the alpha channel” refers to the extra channel with index `alpha_channel`, `sample` is the value after blending, `new_sample` and `old_sample` are the corresponding values from the current and source frame respectively, and `alpha`, `new_alpha` and `old_alpha` are the corresponding values for the alpha channel. If `clamp` is `true`, then for blend modes **kBlend** and **kMulAdd**, values of `new_alpha` are clamped to the interval [0, 1] before blending, and for blend mode **kMul**, the values of `new_sample` are clamped to [0, 1] before blending.

Table F.8 — BlendMode (BlendingInfo.mode)

name	value	meaning
kReplace	0	Each sample is overwritten with the corresponding new sample. `sample = new_sample`
kAdd	1	Each new sample is added to the corresponding previous sample. `sample = old_sample + new_sample`
kBlend	2	Each new sample is alpha-blended over the corresponding previous sample. If the alpha channel has premultiplied semantics (`alpha_associated == true`), then `sample = new_sample + old_sample * (1 − new_alpha)` Otherwise `sample = (new_alpha * new_sample + old_alpha * old_sample * (1 − new_alpha)) / alpha`. The blending on the alpha channel itself always uses the following formula instead: `alpha = old_alpha + new_alpha * (1 − old_alpha)`
kMulAdd	3	Each new sample is multiplied with alpha and added to the corresponding previous sample. `sample = old_sample + new_alpha * new_sample` For the alpha channel itself, the values of the source frame are preserved: `alpha = old_alpha`.
kMul	4	Each new sample is multiplied with the previous sample. `sample = old_sample * new_sample`

`duration` (in units of ticks, see AnimationHeader) is the intended period of time between presenting the current frame and the next one. If `duration` is zero and `!is_last`, the decoder does not present the current frame, but the frame may be composed together with the next frames, for example through blending. In particular, in the case that `metadata.have_animation` is `false`, the decoder returns a single image consisting of the composition of all the zero-duration frames. If `duration` has the value `0xFFFFFFFF`, the decoder presents the next frame as the next page in a multi-page image.

`timecode` indicates the SMPTE timecode of the current frame, or 0. The decoder interprets groups of 8 bits from most-significant to least-significant as hour, minute, second, and frame. If `timecode` is nonzero, it is strictly larger than that of a previous frame with nonzero duration.

The decoder ceases decoding after the current frame if `is_last`.

If `save_before_ct` is `false`, then it is not that case that both `metadata.xyb_encoded == true` and `metadata.colour_encoding.want_icc == true`.

NOTE 2 This implies that a decoder does not need to be able to interpret ICC profiles in order to perform frame blending. It does however need to be able to convert XYB to the colour space of `metadata.colour_encoding` in the non-ICC case, when `save_before_ct == false`.

If `name_len` > 0, then the frame has a name `name`, interpreted as a UTF-8 encoded string.

Finally, `restoration_filter` is defined in J.1.

F.3 TOC

F.3.1 General

The codestream consists of parts called sections. The TOC (Table of Contents) is an array of numbers. Each TOC number indicates the size in bytes of a section. When decoding a section, no more bits are read from the codestream than 8 times the byte size indicated in the TOC; if fewer bits are read, then the remaining bits of the section all have the value zero.

If `num_groups == 1` and `num_passes == 1`, then there is a single TOC entry and a single section containing all frame data structures. Otherwise, there is one entry for each of the following sections, in the order they are listed:

— one section for LfGlobal;

— `num_lf_groups` sections, one per LfGroup (in raster order);

— one section for HfGlobal;

— num_groups * frame_header.passes.num_passes sections, one per PassGroup. The first num_groups PassGroup sections are the groups (in raster order) of the first pass, followed by all groups (in raster order) of the second pass (if present), and so on.

NOTE 1 Some of these sections may be empty in case encoding == **kModular**, i.e. their size in bytes is zero.

The decoder first reads permuted_toc = Bool(). If and only if its value is true, the decoder reads permutation (F.3.2) from a single entropy coded stream with 8 pre-clustered distributions, as specified in C.1, with size equal to the number of TOC entries and skip = 0.

NOTE 2 In case there is a single TOC entry, permuting is useless, but permuted_toc is still signalled.

F.3.2 Decoding permutations

This subclause describes how for a given value of size and skip, the decoder reads a sequence permutation that describes a permutation of size elements of which the first skip elements are not moved.

Let GetContext(x) denote min(7, ceil(log2(x + 1))). The decoder first decodes an integer end, as specified in C.3.3, using DecodeHybridVarLenUint(GetContext(size)). The value end is at most size - skip. Then a sequence lehmer of size elements is produced as follows. It is zero-initialized. For each index i in range [skip, skip + end), the value lehmer[i] is set to DecodeHybridVarLenUint(GetContext(i > skip ? lehmer[i - 1] : 0)); this value is strictly less than size - i.

The decoder then maintains a sequence of elements temp, initially containing the numbers [0, size) in increasing order, and a sequence of elements permutation, initially empty. Then, for each integer i in the range [0, size), the decoder appends to permutation element temp[lehmer[i]], then removes it from temp, leaving the relative order of other elements unchanged. Finally, permutation is the decoded permutation.

F.3.3 Decoding TOC

Before decoding the TOC, but after decoding the TOC permutation, if present, the decoder invokes ZeroPadToByte() (B.2.7).

The decoder reads each TOC entry in order of increasing index via U32(u(10), 1024 + u(14), 17408 + u(22), 4211712 + u(30)).

The decoder then computes an array group_offsets, which has 0 as its first element and subsequent group_offsets[i] are the sum of all TOC entries [0, i).

If permuted_toc, the decoder permutes group_offsets according to permutation, such that group_offsets[i] is what was previously group_offsets[permutation[i]].

After decoding the TOC, the decoder invokes ZeroPadToByte() (B.2.7). Let P be the byte position in the codestream at this point. When decoding a group with index i, the decoder reads from the codestream starting at the byte at position P + group_offsets[i].

Annex G
(normative)

Frame data sections

G.1 LfGlobal

G.1.1 General

The decoder reads the bundles listed in Table G.1.

Table G.1 — LfGlobal bundle

condition	type	name	subclause
kPatches	Patches	`patches`	K.3.1
kSplines	Splines	`splines`	K.4.1
kNoise	NoiseParameters	`noise`	K.5.1
	LfChannelDequantization	`lf_dequant`	G.1.2
`encoding` == **kVarDCT**	Quantizer	`quantizer`	I.2.1
`encoding` == **kVarDCT**	HF Block Context	`hf_block_ctx`	I.2.2
`encoding` == **kVarDCT**	LfChannelCorrelation	`lf_chan_corr`	I.2.3
	GlobalModular	`gmodular`	G.1.3

G.1.2 LF dequantization weights

The decoder reads the bundles listed in Table G.2.

Table G.2 — LfChannelDequantization bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	F16()	1/32	`m_x_lf`
`!all_default`	F16()	1/4	`m_y_lf`
`!all_default`	F16()	1/2	`m_b_lf`

From these values, the LF dequantization weights are computed as follows:

```
m_x_lf_unscaled = m_x_lf / 128;
m_y_lf_unscaled = m_y_lf / 128;
m_b_lf_unscaled = m_b_lf / 128;
```

G.1.3 GlobalModular

First, the decoder reads a Bool() to determine whether a global tree is to be decoded. If `true`, an MA tree is decoded as described in H.4.2.

The decoder then decodes a modular sub-bitstream (Annex H), where the number of channels is computed as follows:

```
  num_channels = num_extra;
  if (frame_header.encoding == kModular) {
    if (!frame_header.do_YCbCr and !metadata.xyb_encoded
        and metadata.colour_encoding.colour_space==kGrey) {
      num_channels += 1;
    } else {
      num_channels += 3;
    }
  }
```

The dimensions correspond to `width` × `height`, except for extra channels with `dim_shift > 0` (as specified in D.3.6) which have dimensions `ceil(width / (1 << dim_shift))` × `ceil(height / (1 << dim_shift))`.

The channel order is:

— If `encoding` == **kModular**: first Grey or (Red, Green, Blue) or (Y', X', B') or (Cb, Y, Cr)

— Then the extra channels (if any), including alpha channel, in ascending order of index.

However, the decoder only decodes the first `nb_meta_channels` channels and any further channels that have a `width` and `height` that are both at most `group_dim`. At that point, it stops decoding. No inverse transforms are applied yet.

NOTE The remaining channels are split into groups and are decoded later (see G.2.3 and G.4.2). If the image is smaller than `group_dim` , the entire image is encoded in the GlobalModular section. Otherwise, the GlobalModular section only encodes global palettes and colour transforms, and all of the actual image data is encoded in modular LF groups and modular groups, unless the Squeeze transform is used. If the Squeeze transform is used (a Haar-like transform that results in a pyramid representation, see H.6.2) then the GlobalModular section does contain partial image data corresponding to (very) downscaled versions of the image, though for large images, the bulk of the pixel data (i.e. Squeeze residuals) is still encoded in groups.

G.2 LfGroup

G.2.1 General

In the clauses related to LfGroup, `width` and `height` refer to the dimensions of the current LF group, and all coordinates are relative to the top-left corner of the current LF group. Table G.3 specifies LfGroup.

Table G.3 — LfGroup

condition	type	name	subclause
`encoding` == **kVarDCT**	LF coefficients	`lf_coeff`	G.2.2
	ModularLfGroup	`mlf_group`	G.2.3
`encoding` == **kVarDCT**	HF metadata	`hf_meta`	G.2.4

G.2.2 LF coefficients

If the **kUseLfFrame** flag in `frame_header` is set, this subclause is skipped and the samples from LFFrame[`frame_header.lf_level`] are used instead of the values that would be computed by this subclause and I.5.2. For the purpose of determining the context when decoding HF coefficients (I.4), the quantized LF coefficients LfQuant are all set to −∞, that is, regardless of `lf_thresholds`, the value of `lf_idx` at the end of the function `BlockContext()` (I.4) is always equal to zero.

The decoder first reads `extra_precision` as a u(2). Next, the decoder reads a Modular sub-bitstream as described in Annex H, to obtain the quantized LF coefficients LfQuant, which consists of three channels with `ceil(height / 8)` rows and `ceil(width / 8)` columns, where the number of rows and columns is optionally right-shifted by one according to `frame_header.jpeg_upsampling`.

Finally the LF is dequantized as specified in I.5.2.

G.2.3 ModularLfGroup

In the partial image decoded from the GlobalModular section, the pixels in the remaining channel data that correspond to the LF group are decoded as another modular image sub-bitstream (Annex H), where the number of channels and their dimensions are derived as follows: for every remaining channel (i.e. not already decoded) in the partially decoded GlobalModular image, if that channel has hshift and vshift both at least 3, then a channel corresponding to the LF group rectangle is added, with the same hshift and vshift as in the GlobalModular image, where the group dimensions and the x,y offsets are right-shifted by hshift (for x and width) and vshift (for y and height).

The decoded modular LF group data is then copied into the partially decoded GlobalModular image in the corresponding positions.

NOTE Since the LF group dimension is at most (8 * group_dim) and hshift and vshift are >= 3, the maximum channel dimensions are group_dim.

G.2.4 HF metadata

The decoder reads nb_blocks = 1 + u(ceil(log2(ceil(width / 8) * ceil(height / 8)))).

Then, the decoder reads a Modular sub-bitstream as described in Annex H, for an image with four channels: the first two channels have ceil(height / 64) rows and ceil(width / 64) columns, and are denoted as XFromY and BFromY (these are the HF colour correlation factors); the third channel has two rows and nb_blocks columns and is denoted as BlockInfo, and the fourth channel has ceil(height / 8) rows and ceil(width / 8) columns and is denoted as Sharpness (it contains parameters for the restoration filter).

The DctSelect and HfMul fields are derived from the first and second rows of BlockInfo. These two fields have ceil(height / 8) rows and ceil(width / 8) columns. They are reconstructed by iterating over the columns of BlockInfo to obtain a varblock transform type type (the sample at the first row) and a quantization multiplier mul (the sample at the second row). The type is a DctSelect sample and is stored at the coordinates of the top-left 8 × 8 rectangle of the varblock. This position is the earliest block in raster order that is not already covered by other varblocks. The positioned varblock is completely contained in the current LF group, does not cross group boundaries, and also does not overlap with already-positioned varblocks. The HfMul sample is stored at the same position and gets the value 1 + mul.

The transform types (defined in I.7) are associated with the numerical values in Table I.1.

G.3 HfGlobal

The decoder reads the structures in Table G.4.

Table G.4 — HfGlobal bundle

condition	type	name	subclause
encoding == **kVarDCT**		Dequantization matrices	I.2.4
encoding == **kVarDCT**		Number of HF decoding presets	I.2.6
encoding == **kVarDCT**	HfPass	hf_pass[num_passes]	I.3

G.4 PassGroup

G.4.1 General

In this clause (and the clauses it refers to), width and height refer to the size of the current group (at most group_dim × group_dim), and all coordinates are relative to the top-left corner of the group. See Table G.5.

Table G.5 — PassGroup bundle

condition	name	subclause
encoding == **kVarDCT**	HF coefficients	I.4
	Modular group data	G.4.2

G.4.2 Modular group data

In the partial image decoded from the GlobalModular and ModularLfGroup sections, the pixels in the remaining channel data that correspond to the group are decoded as another modular image sub-bitstream (Annex H). The values `minshift` and `maxshift` are defined as follows. If this is the first (or only) pass, then `maxshift = 3`, otherwise `maxshift` is equal to the `minshift` of the previous pass. If there is an `n` such that the current pass index is equal to `last_pass[n]`, then `minshift = log2(downsample[n])`, otherwise `minshift = maxshift` (i.e. this pass contains no modular data).

NOTE If there is a non-final pass `i` such that `downsample[k] == 1` and `last_pass[k] == i`, then this pass finalizes the modular data and any following passes will have `minshift == maxshift == 0` so they contain no modular data (though they could still contain VarDCT HF coefficients).

The number of channels and their dimensions are derived as follows: for every remaining channel in the partially decoded GlobalModular image (i.e. it is not a meta-channel, the channel dimensions exceed `group_dim` × `group_dim`, and `hshift < 3` or `vshift < 3`, and the channel has not been already decoded in a previous pass), if that channel has `hshift` and `vshift` such that `minshift <= min(hshift,vshift) < maxshift`, then a channel corresponding to the group rectangle is added, with the same `hshift` and `vshift` as in the GlobalModular image, where the group dimensions and the `x`,`y` offsets are right-shifted by `hshift` (for `x` and `width`) and `vshift` (for `y` and `height`). The decoded modular group data is then copied into the partially decoded GlobalModular image in the corresponding positions.

When all modular groups are decoded, the inverse transforms are applied to the at that point fully decoded GlobalModular image, as specified in H.6.

The reconstructed samples, which at this point are integers, are then interpreted according to the BitDepth (D.3.5), as follows. If `!metadata.xyb_encoded`, then the colour channels (Grey, RGB or CbYCr) are interpreted according to `metadata.bit_depth`. The extra channels are interpreted according to `metadata.ec_info[i].bit_depth`, where `i` is the index of the extra channel.

Annex H
(normative)

Modular

H.1 General

This annex describes the modular image sub-bitstream, which encodes all the pixel data of a frame if `encoding == kModular`. It is also used to encode additional channels (alpha, depth and other extra channels) and auxiliary images in the **kVarDCT** frame encoding. All Modular-encoded sample values have integer values.

The modular image sub-bitstream encodes an image consisting of an arbitrary number N of channels. The dimensions are implicit (i.e. they are signalled or computed elsewhere). In the trivial case where N is zero, the decoder takes no action. The channels are seen as an ordered list: `channel[0]`, ..., `channel[N − 1]`.

Each channel has a width `channel[i].width` and height `channel[i].height`; initially, before transformations are taken into account, all channel dimensions are identical (with the exception of subsampled YCbCr channels and dimension-shifted extra channels, if present, and the HF Metadata encoding as described in G.2.4). Transformations can be applied to the image, which can result in changes to the number of channels and their dimensions. The sub-bitstream starts with an encoding of the series of transformations that was applied, so the decoder can anticipate the corresponding changes in the number of channels and their dimensions (so this information does not need to be explicitly signalled) and can afterwards apply the appropriate inverse transformations.

Channels have a power-of-two horizontal and vertical subsampling factor, denoted by `channel[i].hshift` and `channel[i].vshift`, which are initialized to zero, except in the case of `do_YCbCr` where they are initialized according to the `jpeg_upsampling`, and except for the extra channels, if present, where they are both initialized to the corresponding channel shift value. They can be modified by the transformations (in particular Squeeze). If the channel dimensions correspond to a subsampling of the original image dimensions, this is denoted with positive values for `hshift` and `vshift`, where the horizontal factor is `1 << hshift` and the vertical factor is `1 << vshift`. If the dimensions are not related to the image dimensions, `hshift` and `vshift` are `-1`. The sample in column `x` and row `y` of channel `i` is denoted as `channel[i](x, y)`.

The first `nb_meta_channels` channels are used to store information related to the Palette transformation. Initially, `nb_meta_channels` is set to zero, but transformations can modify this value.

H.2 Image decoding

The decoder reads the fields specified in Table H.1.

Table H.1 — ModularHeader bundle

condition	type	name
	Bool()	`use_global_tree`
	WPHeader (H.5.1)	`wp_params`
	U32(0, 1, 2 + u(4), 18 + u(8))	`nb_transforms`
	TransformInfo (H.6.1)	`transform[nb_transforms]`

The list of channels is initialized according to H.1. Their dimensions and subsampling factors are derived from the series of transforms and their parameters (H.6).

First, if `use_global_tree` is `false`, the decoder reads a MA tree and corresponding clustered distributions as described in H.4.2; otherwise the global MA tree and its clustered distributions are used as decoded from the GlobalModular section (G.1.3). The decoder then starts an entropy-coded stream (C.1) and decodes the data for each channel (in ascending order of index) as specified in H.3, skipping any channels having `width` or `height` zero. Finally, the inverse transformations are applied (from last to first) as described in H.6.

H.3 Channel decoding

The decoding of individual samples of a channel `i` proceeds sequentially in raster order, with `x` the column index of the current sample and `y` its row index. Predictors are computed based on seven previously decoded samples near the current (yet unknown) sample `C`: `NW`, `N`, `NE`, `W`, `NN`, `NEE`, and `WW`, as shown in Table H.2.

Table H.2 — Neighbours used for prediction

	−2	−1	0	+1	+2
−2			NN		
−1		NW	N	NE	NEE
0	WW	W	C		

Edge cases are dealt with as follows, where `channel[i](x, y)` corresponds to the current sample `C`:

```
W = (x > 0 ? channel[i](x − 1, y) : (y > 0 ? channel[i](x, y − 1) : 0);
N = (y > 0 ? channel[i](x, y − 1) : W);
NW = (x > 0 and y > 0 ? channel[i](x − 1, y − 1) : W);
NE = (x + 1 < channel[i].width and y > 0 ? channel[i](x + 1, y − 1) : N);
NN = (y > 1 ? channel[i](x, y − 2) : N);
NEE = (x + 2 < channel[i].width and y > 0 ? channel[i](x + 2, y − 1) : NE);
WW = (x > 1 ? channel[i](x − 2, y) : W);
```

Table H.3 defines the different predictors; the result of predictor number `k` for a (yet to be decoded) sample at coordinates (`x`, `y`) is denoted as `prediction(x, y, k)`.

Table H.3 — Modular predictors

Predictor	Name	Value
0	Zero	`0`
1	West	`W`
2	North	`N`
3	Avg(W,N)	`(W + N) Idiv 2`
4	Select	`abs(N − NW) < abs(W − NW) ? W : N`
5	Gradient	`clamp(W + N − NW, min(W, N), max(W, N))`
6	Self-correcting	`(prediction + 3) >> 3` (see H.5)
7	NorthEast	`NE`
8	NorthWest	`NW`
9	WestWest	`WW`
10	Avg(W,NW)	`(W + NW) Idiv 2`
11	Avg(N,NW)	`(N + NW) Idiv 2`
12	Avg(N,NE)	`(N + NE) Idiv 2`
13	AvgAll	`(6 * N − 2 * NN + 7 * W + WW + NEE + 3 * NE + 8) Idiv 16`

Using the MA tree and distributions D (H.2), channel `i` is reconstructed as follows, where the `dist_multiplier` from C.3.3 is set to the largest channel width amongst all channels that are to be decoded. The function `GetProperties()` is defined in H.4.1 .

```
for (y = 0; y < channel[i].height; y++)
  for (x = 0; x < channel[i].width; x++) {
    properties = GetProperties(i, x, y);
    leaf_node = MA(properties);
    diff = DecodeHybridVarLenUint(leaf_node.ctx);
    diff = UnpackSigned(diff) * leaf_node.multiplier + leaf_node.offset;
    channel[i](x, y) = diff + prediction(x, y, leaf_node.predictor);
  }
```

H.4 Meta-adaptive (MA) context modeling

H.4.1 Meta-adaptive context model

The Meta-Adaptive context model uses a vector of signed 32-bit integer numbers, which are called properties. To determine which context to use, a decision tree is used. The full structure of the tree is known in advance by the decoder. It is explicitly signalled: H.2 describes where and H.4.2 describes how. The inner nodes (decision nodes) of a MA tree contain a test of the form `property[k] > value`. If this test evaluates to `true`, then the left branch is taken, otherwise the right branch is taken. Eventually a leaf node `ln` is reached, which corresponds to a context `ln.ctx` to be used to decode a symbol, a predictor `ln.predictor` to be used, and a multiplier `ln.multiplier` and offset `ln.offset` to apply. The properties used in the MA context model are given in Table H.4. Neighbouring already-decoded samples are defined as in Table H.2.

Table H.4 — Property definitions

Property	Value
0	i (channel index)
1	stream index
2	y
3	x
4	abs(`N`)
5	abs(`W`)
6	`N`
7	`W`
8	`x > 0 ? W - /* (the value of property 9 at position (x - 1, y)) */ : W`
9	`W + N - NW`
10	`W - NW`
11	`NW - N`
12	`N - NE`
13	`N - NN`
14	`W - WW`
15	`max_error` (see H.5.1)

The stream index is defined as follows: for GlobalModular: 0; for LF coefficients: 1 + LF group index; for ModularLfGroup: 1 + `num_lf_groups` + LF group index; for HFMetadata: `1 + 2 * num_lf_groups` + LF group index; for RAW dequantization tables: `1 + 3 * num_lf_groups` + parameters index (see I.2.4); for ModularGroup: `1 + 3 * num_lf_groups + 17 + num_groups *` pass index + group index.

The following code specifies a variable number of additional "previous channel" properties:

```
k = 16;
for (j = i - 1; j >= 0; j--) {
  if (channel[j].width != channel[i].width) continue;
  if (channel[j].height != channel[i].height) continue;
  if (channel[j].hshift != channel[i].hshift) continue;
  if (channel[j].vshift != channel[i].vshift) continue;
  rC = channel[j](x, y);
  rW = (x > 0 ? channel[j](x - 1, y) : 0);
  rN = (y > 0 ? channel[j](x, y - 1) : rW);
  rNW = (x > 0 and y > 0 ? channel[j](x - 1, y - 1) : rW);
  rG = clamp(rW + rN - rNW, min(rW, rN), max(rW, rN));
  property[k++] = abs(rC);
  property[k++] = rC;
  property[k++] = abs(rC - rG);
  property[k++] = rC - rG;
}
```

The function `GetProperties(i, x, y)` returns an array of properties as defined above, where property values that are not set by the above table and code are zero, and values v outside the `int32_t` range are set to `NarrowToI32(v)`.

To find the MA leaf node, the MA tree is traversed, starting at the root node `tree[0]` and for each decision node `d`, testing if `property[d.property] > d.value`, proceeding to the node `tree[d.left_child]` if the test evaluates to `true` and to the node `tree[d.right_child]` otherwise, until a leaf node is reached.

H.4.2 MA tree decoding

The MA tree itself is decoded as follows. The decoder reads 6 pre-clustered distributions from the codestream as specified in C.1, and then decodes the tree as specified by the following code:

```
decode_tree() {
  ctx_id = 0; nodes_left = 1; tree.clear();
  while (nodes_left > 0) {
    nodes_left--;
    property = DecodeHybridVarLenUint(1) - 1;
    if (property >= 0) {
      decision_node.property = property;
      decision_node.value = UnpackSigned(DecodeHybridVarLenUint(0));
      decision_node.left_child = tree.size() + nodes_left + 1;
      decision_node.right_child = tree.size() + nodes_left + 2;
      tree.push_back(decision_node); nodes_left += 2;
    } else {
      leaf_node.ctx = ctx_id++;
      leaf_node.predictor = DecodeHybridVarLenUint(2);
      leaf_node.offset = UnpackSigned(DecodeHybridVarLenUint(3));
      mul_log = DecodeHybridVarLenUint(4);
      mul_bits = DecodeHybridVarLenUint(5);
      leaf_node.multiplier = (mul_bits + 1) << mul_log;
      tree.push_back(leaf_node);
    }
  }
}
```

None of the values of `mul_log` are strictly larger than 30, and none of the values of `mul_bits` are strictly larger than `(1 << (31 - mul_log)) - 2`. At the end of this procedure, `tree.size() <= (1 << 26)`.

`MA(properties)` denotes the leaf node resulting from traversing the MA tree using property values `properties`.

Finally, the decoder reads `(tree.size() + 1) / 2` pre-clustered distributions D as specified in C.1.

H.5 Self-correcting predictor

H.5.1 General

Predictor number 6 is a self-correcting, weighted predictor. For each sample (including samples that use a different predictor), this predictor is invoked as specified in H.5.2. It computes a `prediction` (which is used in H.3) and a `max_error` (which is used in Table H.4), as well as 4 sub-predictor values `subpred[i]`, with i in [0, 4).

After decoding a sample, the decoder then computes `true_err` and `err[i]` for the current sample and stores them for use in predictions of next samples.

The `true_value` corresponds to the decoded sample value. The `true_err` for that sample is calculated as NarrowToI32(`prediction - (true_value << 3)`). The error `err[i]` of sub-predictor `subpred[i]` is computed as follows: `err[i] = (abs(subpred[i] - (true_value << 3)) + 3) >> 3`.

WPHeader (Table H.5) specifies the weight parameters of the self-correcting predictor.

Table H.5 — WPHeader bundle

condition	type	default	name
	Bool()	true	default_wp
!default_wp	u(5)	16	wp_p1
!default_wp	u(5)	10	wp_p2
!default_wp	u(5)	7	wp_p3a
!default_wp	u(5)	7	wp_p3b
!default_wp	u(5)	7	wp_p3c
!default_wp	u(5)	0	wp_p3d
!default_wp	u(5)	0	wp_p3e
!default_wp	u(4)	13	wp_w0
!default_wp	u(4)	12	wp_w1
!default_wp	u(4)	12	wp_w2
!default_wp	u(4)	12	wp_w3

H.5.2 Prediction

The variables N3, NW3, NE3, W3, NN3, and WW3 are computed as the true values of the corresponding samples N, NW, NE, W, NN, WW (cf. Table H.2), left shifted by 3. Edge cases are dealt with as described in H.3. Neighbouring sample symbols at the end of a variable name refer to the corresponding variable for the sample at that location (e.g., true_err_W is the value of true_err for the sample to the left of the current sample; err[i]_N is the value of err[i] for the sample above the current sample).

Edge cases for true_err and err[i] are dealt with as follows:

— if W, N or WW does not exist, the value 0 is used instead;

— if NW or NE does not exist, the value of N is used instead.

For example, at the rightmost border, true_err_NE is interpreted as true_err_N.

The sub-predictors are computed as specified by the following code:

```
subpred[0] = W3 + NE3 - N3;
subpred[1] = N3 - (((true_err_W + true_err_N + true_err_NE) * wp_p1) >> 5);
subpred[2] = W3 - (((true_err_W + true_err_N + true_err_NW) * wp_p2) >> 5);
subpred[3] = N3 - ((true_err_NW * wp_p3a + true_err_N * wp_p3b +
             true_err_NE * wp_p3c + (NN3 - N3) * wp_p3d + (NW3 - W3) * wp_p3e) >> 5);
```

The weights weight[i] for each of the 4 sub-predictions are computed as specified by the following code, based on the err values already computed for sub-predictors of earlier samples.

```
error2weight(err_sum, maxweight) {
  shift = floor(log2(err_sum + 1)) - 5;
  if (shift < 0) shift = 0;
  return 4 + ((maxweight * ((1 << 24) Idiv ((err_sum >> shift) + 1))) >> shift);
}
err_sum[i] = (err[i]_N + err[i]_W + err[i]_NW + err[i]_WW + err[i]_NE) Umod (1 << 32);
if (x == width - 1) err_sum[i] += err[i]_W;
weight[i] = error2weight(err_sum[i], wp_wi);
```

The prediction is then computed based on the sub-predictions and their weight values as specified by the following code:

```
sum_weights = weight[0] + weight[1] + weight[2] + weight[3];
log_weight = floor(log2(sum_weights)) + 1;
for (i = 0; i < 4; i++) weight[i] = weight[i] >> (log_weight - 5);
sum_weights = weight[0] + weight[1] + weight[2] + weight[3];
s = (sum_weights >> 1) - 1;
for (i = 0; i < 4; i++) s += subpred[i] * weight[i];
prediction = s * ((1 << 24) Idiv sum_weights) >> 24;
```

```
// if true_err_N, true_err_W and true_err_NW don't have the same sign
if (((true_err_N ^ true_err_W) | (true_err_N ^ true_err_NW)) <= 0) {
  prediction = clamp(prediction, min(W3, N3, NE3), max(W3, N3, NE3));
}
```

NOTE The integer divisions above can be implemented efficiently using a look-up table because the numerator is always the same (`1 << 24`) and the denominator is an integer between 1 and 64.

The value of `max_error` is computed as specified by the following code:

```
max_error = true_err_W ;
if (abs(true_err_N) > abs(max_error)) max_error = true_err_N;
if (abs(true_err_NW) > abs(max_error)) max_error = true_err_NW;
if (abs(true_err_NE) > abs(max_error)) max_error = true_err_NE;
```

H.6 Transformations

H.6.1 General

Table H.6 lists the three kinds of Modular transforms:

Table H.6 — TransformId

name	value	meaning
kRCT	0	Reversible Colour Transform
kPalette	1	(Delta-)Palette
kSqueeze	2	Modified Haar transform

Table H.7 specifies the TransformInfo bundle. where `tr` is interpreted as above.

Table H.7 — TransformInfo bundle

condition	type	name
	u(2) (TransformId)	`tr`
`tr` != **kSqueeze**	U32(u(3), 8 + u(6), 72 + u(10), 1096 + u(13))	`begin_c`
`tr` == **kRCT**	U32(6, u(2), 2 + u(4), 10 + u(6))	`rct_type`
`tr` == **kPalette**	U32(1, 3, 4, 1 + u(13))	`num_c`
`tr` == **kPalette**	U32(u(8), 256 + u(10), 1280 + u(12), 5376 + u(16))	`nb_colours`
`tr` == **kPalette**	U32(0, 1 + u(8), 257 + u(10), 1281 + u(16))	`nb_deltas`
`tr` == **kPalette**	u(4)	`d_pred`
`tr` == **kSqueeze**	U32(0, 1 + u(4), 9 + u(6), 41 + u(8))	`num_sq`
`tr` == **kSqueeze**	SqueezeParams (H.6.2.1)	`sp[num_sq]`

Table H.8 specifies the transformations.

Table H.8 — Transforms

Name	`nb_meta_channels`	Channel impact	Subclause
kRCT	no change	no change	H.6.3
kPalette	`+ 1` (or `+ 2 − num_c` if `begin_c < nb_meta_channels`)	channels `begin_c` + 1 until `begin_c + num_c − 1` are removed	H.6.4
kSqueeze	depends on parameters (no change with defaults)	depends on parameters	H.6.2

H.6.2 Squeeze

H.6.2.1 Parameters

Table H.9 — SqueezeParams bundle

condition	type	name
	Bool()	horizontal
	Bool()	in_place
	U32(u(3), 8 + u(6), 72 + u(10), 1096 + u(13))	begin_c
	U32(1, 2, 3, 4 + u(4))	num_c

The squeeze transform consists of a series of horizontal and vertical squeeze steps. The sequence of squeeze steps to be applied is defined by sp, which is an array of SqueezeParams (see Table H.9); if the array is empty, default parameters are used which are derived from the image dimensions and number of channels.

For every squeeze step sp[i], the sp[i].num_c input channels starting at position sp[i].begin_c are replaced by the squeezed channels, and the residual channels are inserted either right after the squeezed channels (if sp[i].in_place == true), or at the end of the channel list (otherwise). To determine the new list of channels (i.e. when interpreting the transformation description, before channel data decoding starts), the steps are applied in the order in which they are specified. After the channel data has been decoded, to apply the inverse transformation, the steps are applied in reverse order.

Let begin = sp[i].begin_c and end = begin + sp[i].num_c - 1. The channel list is modified as specified by the following code:

```
for (i = 0; i < sp.size(); i++) {
  r = sp[i].in_place ? end + 1 : channel.size();
  if (begin < nb_meta_channels) {
    /* sp[i].in_place is true */
    /* end < nb_meta_channels */
    nb_meta_channels += sp[i].num_c;
  }
  for (c = begin; c <= end; c++) {
    w = channel[c].width;
    h = channel[c].height;
    /* w > 0 and h > 0 */
    if (sp[i].horizontal) {
      channel[c].width = (w + 1) Idiv 2;
      if (channel[c].hshift >= 0) channel[c].hshift++;
      residu = channel[c].copy();
      residu.width = w Idiv 2;
    } else {
      channel[c].height = (h + 1) Idiv 2;
      if (channel[c].vshift >= 0) channel[c].vshift++;
      residu = channel[c].copy();
      residu.height = h Idiv 2;
    }
    /* Insert residu into channel at index r + c - begin */
  }
}
```

The inverse transform is specified by the following code:

```
for (i = sp.size() - 1; i >= 0; i--) {
  r = sp[i].in_place ? end + 1 : channel.size() + begin - end - 1;
  for (c = begin; c <= end; c++) {
    Channel output = channel[c].copy();
    if (sp[i].horizontal) {
      output.width += channel[r].width;
      horiz_isqueeze(channel[c], channel[r], output);
    } else {
      output.height += channel[r].height;
      vert_isqueeze(channel[c], channel[r], output);
    }
    channel[c] = output;
    /* Remove the channel with index r */
  }
}
```

The default parameters (the case when `sp.size() == 0`) are specified by the following code:

```
first = nb_meta_channels; count = channel.size() - first;
w = channel[first].width; h = channel[first].height;
if (count > 2 and channel[first + 1].width == w and channel[first + 1].height == h) {
  param.begin_c = first + 1; param.num_c = 2; param.in_place = false;
  param.horizontal = true; sp.push_back(param);
  param.horizontal = false; sp.push_back(param);
}
param.begin_c = first; param.num_c = count; param.in_place = true;
if (h >= w and h > 8) { param.horizontal = false; sp.push_back(param); h = (h + 1) Idiv 2; }
while (w > 8 or h > 8) {
  if (w > 8) { param.horizontal = true; sp.push_back(param); w = (w + 1) Idiv 2; }
  if (h > 8) { param.horizontal = false; sp.push_back(param); h = (h + 1) Idiv 2; }
}
```

H.6.2.2 Horizontal inverse squeeze step

This step takes two input channels of sizes `W1` × `H` and `W2` × `H`, and replaces them with one output channel of size `(W1 + W2)` × `H`. Either `W1 == W2` or `W1 == W2 + 1`. The output is reconstructed as follows:

```
horiz_isqueeze(input_1, input_2, output) {
  for (y = 0; y < H; y++) {
    for (x = 0; x < W2; x++) {
      avg = input_1(x, y); residu = input_2(x, y);
      next_avg = (x + 1 < W1 ? input_1(x + 1, y) : avg);
      left = (x > 0 ? output((x << 1 ) - 1, y) : avg);
      diff = residu + tendency(left, avg, next_avg);
      first = avg + diff Idiv 2;
      output(2 * x, y) = first;
      output(2 * x + 1, y) = first - diff;
    }
    if (W1 > W2) output(2 * W2) = input_1(W2);
  }
}
```

The tendency function is specified by the following code:

```
tendency(A, B, C) {
  if (A >= B and B >= C) {
    X = (4 * A - 3 * C - B + 6) Idiv 12;
    if (X - (X & 1) > 2 * (A - B)) X = 2 * (A - B) + 1;
    if (X + (X & 1) > 2 * (B - C)) X = 2 * (B - C);
    return X;
  } else if (A <= B and B <= C) {
    X = (4 * A - 3 * C - B - 6) Idiv 12;
    if (X + (X & 1) < 2 * (A - B)) X = 2 * (A - B) - 1;
    if (X - (X & 1) < 2 * (B - C)) X = 2 * (B - C);
    return X;
  } else return 0;
}
```

H.6.2.3 Vertical inverse squeeze step

This step takes two input channels of sizes `W` × `H1` and `W` × `H2`, and replaces them with one output channel of size `W` × `(H1 + H2)`. Either `H1 == H2` or `H1 == H2 + 1`. The output is reconstructed as follows:

```
vert_isqueeze(input_1, input_2, output) {
  for (y = 0; y < H2; y++) {
    for (x = 0; x < W; x++) {
      avg = input_1(x, y); residu = input_2(x, y);
      next_avg = (y + 1 < H1 ? input_1(x, y + 1) : avg);
      top = (y > 0 ? output(x, (y << 1) - 1) : avg);
      diff = residu + tendency(top, avg, next_avg);
      first = avg + diff Idiv 2;
      output(x,2 * y) = first;
      output(x,2 * y + 1) = first - diff;
    }
  }
  if (H1 > H2)
    for (x = 0; x < W; x++) output(x, 2 * H2) = input_1(x, H2);
}
```

H.6.3 RCT (reversible colour transform)

This transformation operates on three channels starting with begin_c. These channels have the same dimensions. Either begin_c >= nb_meta_channels and begin_c + 3 <= channel.size(), or begin_c + 3 <= nb_meta_channels. For every pixel position in these channels, the values (A, B, C) are replaced by (V[0], V[1], V[2]) values, as follows:

```
/* rct_type < 42 */
permutation = rct_type Idiv 7;
type = rct_type Umod 7;
if (type == 6) {        // YCoCg
  tmp = A - (C >> 1);
  E = C + tmp;
  F = tmp - (B >> 1);
  D = F + B;
} else {
  if (type & 1) C = C + A;
  if ((type >> 1) == 1) B = B + A;
  if ((type >> 1) == 2) B = B + ((A + C) >> 1);
  D = A; E = B; F = C;
}
V[permutation Umod 3] = D;
V[(permutation+1+(permutation Idiv 3)) Umod 3] = E;
V[(permutation+2-(permutation Idiv 3)) Umod 3] = F;
```

EXAMPLE If rct_type == 10, then pixels (G, B', R') are replaced by (R' + G, G, B' + G).

H.6.4 Palette

Let end_c = begin_c + num_c - 1. When updating the channel list as described in H.2, channels begin_c to end_c, which all have the same dimensions, are replaced with two new channels:

— one meta-channel, inserted at the beginning of the channel list and has dimensions width = nb_colours and height = num_c and hshift = vshift = -1. This channel represents the colours or deltas of the palette.

— one channel (at the same position in the channel list as the original channels, same dimensions) which contains palette indices.

If begin_c < nb_meta_channels, then end_c < nb_meta_channels is true and nb_meta_channels += 2 - num_c. Otherwise, nb_meta_channels += 1.

The decoder restores the original pixels as specified by the following code:

```
kDeltaPalette[72][3] = {
 {0, 0, 0}, {4, 4, 4}, {11, 0, 0}, {0, 0, -13}, {0, -12, 0}, {-10, -10, -10},
 {-18, -18, -18}, {-27, -27, -27}, {-18, -18, 0}, {0, 0, -32}, {-32, 0, 0}, {-37, -37, -37},
 {0, -32, -32}, {24, 24, 45}, {50, 50, 50}, {-45, -24, -24}, {-24, -45, -45}, {0, -24, -24},
 {-34, -34, 0}, {-24, 0, -24}, {-45, -45, -24}, {64, 64, 64}, {-32, 0, -32}, {0, -32, 0},
 {-32, 0, 32}, {-24, -45, -24}, {45, 24, 45}, {24, -24, -45}, {-45, -24, 24}, {80, 80, 80},
 {64, 0, 0}, {0, 0, -64}, {0, -64, -64}, {-24, -24, 45}, {96, 96, 96}, {64, 64, 0},
 {45, -24, -24}, {34, -34, 0}, {112, 112, 112}, {24, -45, -45}, {45, 45, -24}, {0, -32, 32},
 {24, -24, 45}, {0, 96, 96}, {45, -24, 24}, {24, -45, -24}, {-24, -45, 24}, {0, -64, 0},
 {96, 0, 0}, {128, 128, 128}, {64, 0, 64}, {144, 144, 144}, {96, 96, 0}, {-36, -36, 36},
 {45, -24, -45}, {45, -45, -24}, {0, 0, -96}, {0, 128, 128}, {0, 96, 0}, {45, 24, -45},
 {-128, 0, 0}, {24, -45, 24}, {-45, 24, -45}, {64, 0, -64}, {64, -64, -64}, {96, 0, 96},
 {45, -45, 24}, {24, 45, -45}, {64, 64, -64}, {128, 128, 0}, {0, 0, -128}, {-24, 45, -45} };

first = begin_c + 1; last = end_c + 1; bitdepth = metadata.bit_depth.bits_per_sample;
for (i = first + 1; i <= last; i++) /* Insert a copy of channel[first] at index i */;
for (c = 0; c < num_c; c++)
  for (y = 0; y < channel[first].height; y++)
```

```
    for (x = 0; x < channel[first].width; x++) {
      index = channel[first + c](x, y); is_delta = (index < nb_deltas);
      if (index >= 0 and index < nb_colours) { value = channel[0](index, c);
      } else if (index >= nb_colours) {
        index -= nb_colours;
        if (index < 64) {
          value = ((index >> (2 * c)) Umod 4) * ((1 << bitdepth) - 1) / 4
                  + (1 << max(0, bitdepth - 3));
        } else {
          index -= 64;
          for (i = 0; i < c; i++) index = index Idiv 5;
          value = (index Umod 5) * ((1 << bitdepth) - 1) / 4;
        }
      } else if (c < 3) {
        index = (-index - 1) Umod 143;
        value = kDeltaPalette[(index + 1) >> 1][c];
        if (index & 1 == 0) value = -value;
        if (bitdepth > 8) value <<= min(bitdepth, 24) - 8;
      } else value = 0;
      channel[first + c](x, y) = value;
      if (is_delta) channel[first + c](x, y) += prediction(x, y, d_pred);
    }
/* Remove channel 0 */;
```

Annex I
(normative)

VarDCT

I.1 Transform types

Table I.1 specifies the different transform types that are used in the **kVarDCT** encoding, and the numerical values that are used to represent them in `DctSelect` (c.f. G.2.4).

Table I.1 — Transform type values

Transform type	Numerical value	Dimensions in `DctSelect`
DCT8×8	0	1×1
Hornuss	1	1×1
DCT2×2, DCT4×4	2, 3	1×1
DCT16×16	4	2×2
DCT32×32	5	4×4
DCT16×8	6	2×1
DCT8×16	7	1×2
DCT32×8	8	4×1
DCT8×32	9	1×4
DCT32×16	10	4×2
DCT16×32	11	2×4
DCT4×8, DCT8×4	12, 13	1×1
AFV0 – AFV3	14 – 17	1×1
DCT64×64	18	8×8
DCT64×32	19	8×4
DCT32×64	20	4×8
DCT128×128	21	16×16
DCT128×64	22	16×8
DCT64×128	23	8×16
DCT256×256	24	32×32
DCT256×128	25	32×16
DCT128×256	26	16×32

I.2 Decoding quantization and decorrelation parameters

I.2.1 Quantizer

Table I.2 specifies the Quantizer bundle.

Table I.2 — Quantizer bundle

condition	type	name
	U32(1 + u(11), 2049 + u(11), 4097 + u(12), 8193 + u(16))	`global_scale`
	U32(16, 1 + u(5), 1 + u(8), 1 + u(16))	`quant_lf`

The LF dequantization factors `mXDC`, `mYDC` and `mBDC` are computed as specified by the following code:

```
mXDC = (1 << 16) * m_x_lf_unscaled / (global_scale * quant_lf);
mYDC = (1 << 16) * m_y_lf_unscaled / (global_scale * quant_lf);
mBDC = (1 << 16) * m_b_lf_unscaled / (global_scale * quant_lf);
```

I.2.2 HF block context decoding

The decoder reads a description of the block context model for HF as specified by the following code, where `ReadThreshold()` denotes U32(u(4), 16 + u(8), 272 + u(16), 65808 + u(32)) and `ReadBlockCtxMap()` reads a clustering map as in C.2.2.

```
/* initialize qf_thresholds to an empty vector */;
/* initialize lf_thresholds to an array of 3 empty vectors */;
if (u(1)) block_ctx_map = {0, 1, 2, 2, 3, 3, 4, 5, 6, 6, 6, 6, 6,
                           7, 8, 9, 9, 10, 11, 12, 13, 14, 14, 14, 14, 14,
                           7, 8, 9, 9, 10, 11, 12, 13, 14, 14, 14, 14, 14};
else {
  for (i = 0; i < 3; i++) {
    nb_lf_thr[i] = u(4);
    for (j = 0; j < nb_lf_thr[i]; j++) {
      t = UnpackSigned(ReadThreshold());
      lf_thresholds[i].push_back(t);
    }
  }
  nb_qf_thr = u(4);
  for (i = 0; i < nb_qf_thr; i++)
    qf_thresholds.push_back(1 + U32(u(2), 4 + u(3), 12 + u(5), 44 + u(8))));
  bsize = 39 * (nb_qf_thr + 1) * (nb_lf_thr[0] + 1) * (nb_lf_thr[1] + 1) * (nb_lf_thr[2] + 1);
  block_ctx_map = ReadBlockCtxMap();
  /* num_dist = bsize <= 39 * 64 and the resulting num_clusters <= 16 */;
}
```

I.2.3 LF channel correlation factors

Table I.3 specifies the LfChannelCorrelation bundle.

Table I.3 — LfChannelCorrelation bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	U32(84,256, 2 + u(8), 258 + u(16))	84	`colour_factor`
`!all_default`	F16()	0.0	`base_correlation_x`
`!all_default`	F16()	1.0	`base_correlation_b`
`!all_default`	u(8)	128	`x_factor_lf`
`!all_default`	u(8)	128	`b_factor_lf`

I.2.4 Dequantization matrices

The dequantization matrices are used as multipliers for the HF coefficients, as specified in I.5.3. They are defined by the channel, the transform type and the index of the coefficient inside the varblock. The parameters that define the dequantization matrices are read from the stream as follows. First, the decoder reads a Bool(). If this is `true`, all matrices have their default encoding as specified in I.2.5. Otherwise, the decoder reads 17 sets of parameters from the codestream, in ascending order. Each set of parameters is used for the values of the `DctSelect` field specified in Table I.4.

Table I.4 — DctSelect values for dequantization matrices

Parameters index	DctSelect values	Matrix size (rows columns)
0	DCT	8×8
1	Hornuss	8×8
2	DCT2×2	8×8
3	DCT4×4	8×8
4	DCT16×16	16×16
5	DCT32×32	32×32
6	DCT16×8, DCT8×16	8×16
7	DCT32×8, DCT8×32	8×32
8	DCT16×32, DCT32×16	16×32
9	DCT4×8, DCT8×4	8×8
10	AFV0, AFV1, AFV2, AFV3	8×8
11	DCT64×64	64×64
12	DCT32×64, DCT64×32	32×64
13	DCT128×128	128×128
14	DCT64×128, DCT128×64	64×128
15	DCT256×256	256×256
16	DCT128×256, DCT256×128	128×256

Each parameter in this subclause is read using F16() (B.2.4) unless otherwise specified. For each matrix, the `encoding_mode` is read as a u(3) and interpreted per Table I.5.

Table I.5 — Encoding mode and valid indices

`encoding_mode`	Name	Valid index
0	Library	all
1	Hornuss	0,1,2,3,9,10
2	DCT2	0,1,2,3,9,10
3	DCT4	0,1,2,3,9,10
4	DCT4x8	0,1,2,3,9,10
5	AFV	0,1,2,3,9,10
6	DCT	all
7	RAW	all

The `encoding_mode` read for a given parameter index is such that the given index is specified as a valid index for that encoding in Table F.1. After reading the `encoding_mode`, the corresponding parameters are read as specified by the following code, where the modular sub-bitstream refers to Annex H and `SetDefaultMode()` sets `params`, `dct_params`, `dct4x4_params`, and `encoding_mode` to the default values from I.2.5.

```
ReadDctParams() {
  num_params = u(4) + 1;
  vals = /* read 3 x num_params matrix in raster order */;
  for (i = 0; i < 3; i++) vals(i, 0) *= 64;
  return vals;
}
if (encoding_mode == Library) {
  SetDefaultMode();
} else if (encoding_mode == Hornuss) {
  params = /* read 3 x 3 matrix in raster order, multiply elements by 64 */;
} else if (encoding_mode == DCT2) {
  params = /* read 3 x 6 matrix in raster order, multiply elements by 64 */;
} else if (encoding_mode == DCT4) {
  params = /* read 3 x 2 matrix in raster order, multiply elements by 64 */;
  dct_params = ReadDctParams();
} else if (encoding_mode == DCT) {
  dct_params = ReadDctParams();
} else if (encoding_mode == RAW) {
  params.denominator = F16();
  params = /* read a 3-channel image from a modular sub-bitstream
              of the same shape as the required quant matrix */;
} else if (encoding_mode == DCT4x8) {
  params = /* read 3 x 1 matrix in raster order */;
  dct_params = ReadDctParams();
} else if (encoding_mode == AFV) {
  params = /* read 3 x 9 matrix in raster order */;
  for (i = 0; i < 3; i++) for (j = 0; j < 6; j++) params(i, j) *= 64;
  dct_params = ReadDctParams();
  dct4x4_params = ReadDctParams();
}
```

The dequantization matrices are computed for each possible value of the `DctSelect` field as the inverse of the weights matrix, that only depends on the corresponding parameters as specified in Table I.4 and is computed per channel.

A weights matrix of dimensions X × Y is computed as specified by `GetDCTQuantWeights()` in the following code, given an array of parameters `params`.

```
Interpolate(pos, max, bands) {
  if (bands.size() == 1) return bands[0];
  scaled_pos = pos * (bands.size() - 1) / max;
  scaled_index = floor(scaled_pos);
  frac_index = scaled_pos - scaled_index;
  A = bands[scaled_index];
  B = bands[scaled_index + 1];
  interpolated_value = A * pow(B / A, frac_index);
  return interpolated_value;
}

Mult(v) { if (v > 0) return 1 + v; else return 1 / (1 - v); }
GetDCTQuantWeights(params) {
  bands.clear();
  bands.push_back(params[0]);
  for (i = 1; i < params.size(); i++) {
    bands.push_back(bands[i - 1] * Mult(params[i]));
    /*  bands[i] > 0  */
  }
  for (y = 0; y < Y; y++)
    for (x = 0; x < X ; x++) {
      dx = x / (X - 1);
      dy = y / (Y - 1);
      distance = sqrt(dx*dx + dy*dy);
      weight = Interpolate(distance, sqrt(2) + 1e-6, bands);
      weights(x, y) = weight;
    }
  }
  return weights;
}
```

For encoding mode DCT, the weights matrix for channel c is the matrix of the correct size computed using `GetDctQuantWeights`, using row c of `dct_params` as an input.

For encoding mode DCT4, the weights matrix for channel c are obtained by copying into position (x, y) the value in position (x Idiv 2, y Idiv 2) in the 4 x 4 matrix computed by `GetDctQuantWeights` using as an input row c of `dct_params`; coefficients (0,1) and (1,0) are divided by `params(c, 0)`, and the (1,1) coefficient is divided by `params(c, 1)`.

For encoding mode DCT2, `params(c, i)` are copied in position (x, y) as follows, where rectangles are defined by their top left and bottom right corners, and *symmetric* refers to the rectangle defined by the same points but with swapped x and y coordinates:

— `i == 0`: positions (0,1) and (1, 0)

— `i == 1`: position (1,1)

— `i == 2`: all positions in the rectangle `((2,0), (4,2))`, and symmetric

— `i == 3`: all positions in the rectangle `((2,2), (4,4))`

— `i == 4`: all positions in the rectangle `((4,0), (8,4))`, and symmetric

— `i == 5`: all positions in the rectangle `((4,4), (8,8))`

For encoding mode Hornuss, coefficient (1,1) is equal to params(c, 2). Coefficients (0,1) and (1,0) are equal to params(c, 1), and all other coefficients to params(c, 0). Coefficient (0,0) is 1.

For encoding mode DCT4x8, the weights matrix is obtained by copying into position (x, y) the value in position (x, y Idiv 2) in the 4 × 8 matrix computed by `GetDctQuantWeights` using as an input row c of `dct_params`; coefficient (0,1) is then divided by `params(c, 0)`.

For encoding mode AFV and channel c, the decoder obtains weights as specified by the following code:

```
freqs = {0, 0, 0.8517778890324296, 5.37778436506804, 0, 0, 4.734747904497923,
5.449245381693219, 1.6598270267479331, 4, 7.275749096817861, 10.423227632456525,
2.662932286148962,  7.630657783650829, 8.962388608184032, 12.97166202570235};
weights4x8 = /* 4 x 8 matrix produced by GetDctQuantWeights using row c of dct_params */
weights4x4 = /* 4 x 4 matrix produced by GetDctQuantWeights using row c of dct4x4_params */
lo = 0.8517778890324296; hi = 12.97166202570235;
bands[0] = params(c, 5);
/*  bands[0] >= 0  */
for (i = 1; i < 4; i++) {
  bands[i] = bands[i - 1] * Mult(params(c, i + 5));
  /*  bands[i] >= 0  */
}
weights(0, 0) = 1; weights(0, 1) = params(c, 0); weights(1, 0) = params(c, 1);
weights(0, 2) = params(c, 2); weights(2, 0) = params(c, 3); weights(2, 2) = params(c, 4);
for (y = 0; y < 4; y++)
  for (x = 0; x < 4; x++) {
     if (x < 2 and y < 2) continue;
     val = Interpolate(freqs[y * 4 + x] - lo, hi - lo + 1e-6, bands);
     weights(2 * y, 2 * x) = val;
   }
for (y = 0; y < 4; y++)
  for (x = 0; x < 8; x++) {
    if (x == 0 and y == 0) continue;
    weights(x, 2 * y + 1) = weights4x8(x, y);
  }
for (y = 0; y < 4; y++)
  for (x = 0; x < 4; x++) {
    if (x == 0 and y == 0) continue;
    weights(2 * x + 1, 2 * y) = weights4x4(x, y);
  }
```

For encoding mode RAW, the dequantization matrices are equal to the `params` matrices multiplied by `params.denominator`.

For all other encoding modes, the dequantization matrices for channel c are the element-wise reciprocals of the weights matrix computed for channel c. None of the resulting values are non-positive or infinity.

I.2.5 Default values for each dequantization matrix

The dequantization matrix encodings defined in Table I.6 are used as the default encodings for each kind of dequantization matrix.

Table I.6 — Default matrix parameters for each DctSelect

DctSelect	mode	dct_params, params
DCT8×8	DCT	{ {3150.0, 0.0, −0.4, −0.4, −0.4, −2.0}, {560.0, 0.0, −0.3, −0.3, −0.3, −0.3}, {512.0, −2.0, −1.0, 0.0, −1.0, −2.0} }, {}
Hornuss	Hornuss	{}, { {280.0, 3160.0, 3160.0}, {60.0, 864.0, 864.0}, {18.0, 200.0, 200.0} }
DCT2×2	DCT2	{}, { {3840.0, 2560.0, 1280.0, 640.0, 480.0, 300.0}, {960.0, 640.0, 320.0, 180.0, 140.0, 120.0}, {640.0, 320.0, 128.0, 64.0, 32.0, 16.0} }
DCT4×4	DCT4	`dct4x4_params`, { {1.0, 1.0}, {1.0, 1.0}, {1.0, 1.0} }
DCT16×16	DCT	{ {8996.8725711814115328, −1.3000777393353804, −0.49424529824571225, −0.43909377445710344З, −0.6350101832695744, −0.90177264050827612, −1.6162099239887414}, {3191.48366296844234752, −0.67424582104194355, −0.80745813428471001, −0.44925837484843441, −0.35865440981033403, −0.31322389111877305, −0.37615025315725483}, {1157.50408145487200256, −2.0531423165804414, −1.4, −0.50687130033378396, −0.42708730624733904, −1.4856834539296244, −4.9209142884401604} }, {}
DCT32×32	DCT	{ {15718.40830982518931456, −1.025, −0.98, −0.9012, −0.4, −0.48819395464, −0.421064, −0.27}, {7305.7636810695983104, −0.8041958212306401, −0.7633036457487539, −0.55660379990111464, −0.49785304658857626, −0.43699592683512467, −0.40180866526242109, −0.27321683125358037}, {3803.53173721215041536, −3.060733579805728, −2.0413270132490346, −2.0235650159727417, −0.5495389509954993, −0.4, −0.4, −0.3} }, {}
DCT16×8, DCT8×16	DCT	{ {7240.7734393502, −0.7, −0.7, −0.2, −0.2, −0.2, −0.5}, {1448.15468787004, −0.5, −0.5, −0.5, −0.2, −0.2, −0.2}, {506.854140754517, −1.4, −0.2, −0.5, −0.5, −1.5, −3.6} }, {}
DCT32×8, DCT8×32	DCT	{ {16283.2494710648897, −1.7812845336559429, −1.6309059012653515, −1.0382179034313539, −0.85, −0.7, −0.9, −1.2360638576849587}, {5089.15750884921511936, −0.320049391452786891, −0.35362849922161446, −0.30340000000000003, −0.61, −0.5, −0.5, −0.6}, {3397.77603275308720128, −0.321327362693153371, −0.34507619223117997, −0.70340000000000003, −0.9, −1.0, −1.0, −1.1754605576265209} }, {}
DCT16×32, DCT32×16	DCT	{ {13844.97076442300573, −0.97113799999999995, −0.658, −0.42026, −0.22712, −0.2206, −0.226, −0.6}, {4798.964084220744293, −0.61125308982767057, −0.83770786552491361, −0.79014862079498627, −0.2692727459704829, −0.38272769465388551, −0.22924222653091453, −0.20719098826199578}, {1807.236946760964614, −1.2, −1.2, −0.7, −0.7, −0.7, −0.4, −0.5} }, {}
DCT4×8, DCT8×4	DCT4x8	`dct4x8_params`, {1.0, 1.0, 1.0}
AFV0, AFV1, AFV2, AFV3	AFV	`dct4x8_params`, { {3072, 3072, 256, 256, 256, 414, 0.0, 0.0, 0.0}, {1024, 1024, 50.0, 50.0, 50.0, 58, 0.0, 0.0, 0.0}, {384, 384, 12.0, 12.0, 12.0, 22, −0.25, −0.25, −0.25} }
DCT64×64	DCT	{ {23966.1665298448605, SeqA}, {8380.19148390090414, SeqB}, {4493.02378009847706, SeqC} },{}
DCT32×64, DCT64×32	DCT	{ {15358.89804933239925, SeqA}, {5597.360516150652990, SeqB}, {2919.961618960011210, SeqC} }, {}
DCT128×128	DCT	{ {47932.3330596897210, SeqA}, {16760.38296780180828, SeqB}, {8986.04756019695412, SeqC} }, {}
DCT64×128, DCT128×64	DCT	{ {30717.796098664792, SeqA}, {11194.72103230130598, SeqB}, {5839.92323792002242, SeqC} }, {}
DCT256×256	DCT	{ {95864.6661193794420, SeqA}, {33520.76593560361656, SeqB}, {17972.09512039390824, SeqC} }, {}
DCT128×256, DCT256×128	DCT	{ {61435.5921973295970, SeqA}, {24209.44206460261196, SeqB}, {12979.84647584004484, SeqC} }, {}

In the above table, the following abbreviations denote a sequence of numbers: SeqA is the sequence −1.025, −0.78, −0.65012, −0.19041574084286472, −0.20819395464, −0.421064, −0.32733845535848671; SeqB is the sequence −0.3041958212306401, −0.3633036457487539, −0.35660379990111464, −0.3443074455424403, −0.33699592683512467, −0.30180866526242109, −0.27321683125358037, SeqC is the sequence −1.2, −1.2, −0.8, −0.7, −0.7, −0.4, −0.5. Furthermore, `dct4x8_params` is { {2198.050556016380522, −0.96269623020744692, −0.76194253026666783, −0.6551140670773547}, {764.3655248643528689, −0.92630200888366945, −0.9675229603596517, −0.27845290869168118}, {527.107573587542228,

−1.4594385811273854, −1.450082094097871593, −1.5843722511996204} } and `dct4x4_params` is { {2200, 0, 0, 0}, {392, 0, 0, 0}, {112, −0.25, −0.25, −0.5} }.

I.2.6 Number of HF decoding presets

The decoder reads `num_hf_presets` as `u(ceil(log2(num_groups))) + 1`. The number of pre-clustered distributions for HF coefficients depends on `num_hf_presets`.

NOTE The number of bits used to represent this number is not a constant because the set of histograms to be used is decided on a per-group basis.

I.3 HfPass

I.3.1 HF coefficient order

The decoder first reads `used_orders` as `U32(0x5F, 0x13, 0x00, u(13))`. If `used_orders != 0`, it reads 8 pre-clustered distributions as specified in C.1. It then reads HF coefficient orders `order[p][b][c]` as specified by the code below, where `p` is the index of the current pass, `b` is an Order ID (see Table I.7), `c` is a component index, and `natural_coeff_order[b]` is the natural coefficient order for Order ID `b`, as specified in I.3.2.

```
for (b = 0; b < 13; b++)
 for (c = 0; c < 3; c++)
  if ((used_orders & (1 << b)) != 0) {
    nat_ord_perm = DecodePermutation(b);
    for /* each coefficient index i */
      order[p][b][c][i] = natural_coeff_order[b][nat_ord_perm[i]];
  } else {
    for /* each coefficient index i */
      order[p][b][c][i] = natural_coeff_order[b][i];
  }
```

`DecodePermutation(b)` is defined as follows. The decoder reads a permutation `nat_ord_perm` from a *single* stream (shared during the above loop) as specified in F.3.2, where `size` is the number of coefficients covered by transforms with Order ID `b` (so `size == natural_coeff_order[b].size()`) and `skip = size / 64`.

I.3.2 Natural ordering of the DCT coefficients

For every `DctSelect` value (Hornuss, DCT2×2, etc), the natural order of the coefficients is computed as follows. The varblock size (`bwidth`, `bheight`) for a `DctSelect` value with name "DCTN×M" is `bwidth = max(8, max(N, M))` and `bheight = max(8, min(N, M))`. The varblock size for all other transforms is `bwidth = bheight = 8`. The natural ordering of the DCT coefficients is defined as a vector order of cell positions (x, y) between (0, 0) and (`bwidth`, `bheight`), described below. The number of elements in the vector order is therefore `bwidth * bheight`, and the vector is defined as the elements of LLF in their original order followed by the elements of HF also in their original order. LLF is a vector of lower frequency coefficients, containing cells (x, y) with `x < bwidth / 8` and `y < bheight / 8`. The cells (x, y) that do not satisfy this condition belong to the higher frequencies vector HF.

The rest of this subclause specifies how to order the elements within each of the arrays LLF and HF. The pairs (x, y) in the LLF vector is sorted in ascending order according to the value `y * bwidth / 8 + x`. For the pairs (x, y) in the HF vector, the decoder first computes the value of the variables `key1` and `key2` as specified by the following code:

```
cx = bwidth / 8; cy = bheight / 8;
scaled_x = x * max(cx, cy) / cx;
scaled_y = y * max(cx, cy) / cy;
key1 = scaled_x + scaled_y;
key2 = scaled_x - scaled_y;
if (key1 Umod 2 == 1) key2 = -key2;
```

The decoder sorts the (x, y) pairs on the vector HF in ascending order according to the value `key1`. In case of a tie, the decoder also sorts in ascending order according to the value `key2`.

The *order ID* is defined based on the `DctSelect` as defined in Table I.7.

Table I.7 — Order ID for DctSelect values

Order ID	DctSelect
0	DCT8×8
1	Hornuss, DCT2×2, DCT4×4, DCT4×8, DCT8×4, AFV0, AFV1, AFV2, AFV3
2	DCT16×16
3	DCT32×32
4	DCT16×8, DCT8×16
5	DCT32×8, DCT8×32
6	DCT32×16, DCT16×32
7	DCT64×64
8	DCT32×64, DCT64×32
9	DCT128×128
10	DCT64×128, DCT128×64
11	DCT256×256
12	DCT128×256, DCT256×128

I.3.3 HF coefficient histograms

Let `nb_block_ctx` be equal to `max(block_ctx_map) + 1`. The decoder reads a histogram with `495 * num_hf_presets * nb_block_ctx` pre-clustered distributions D from the codestream as specified in C.1.

I.4 Decoding of quantized HF coefficients

The decoder reads `hfp = u(ceil(log2(num_hf_presets)))`, which indicates the offset in the histogram, which is given by `offset = 495 * nb_block_ctx * hfp`. These are chosen from the `num_hf_presets` possibilities of the current *pass*.

To compute context, the decoder uses the procedures and constants in the following code, where `c` is the current channel (with 0=X, 1=Y, 2=B), `s` is the Order ID (see Table I.7) of the `DctSelect` value and `qf` is the `HfMul` value for the current varblock, and `qdc[3]` are the quantized LF values of LfQuant (G.2.2) corresponding to (the top-left 8×8 block within) the current varblock (taking into account `jpeg_upsampling` if needed). The lists of thresholds `qf_thresholds` and `lf_thresholds[3]`, and `block_ctx_map` are as decoded in LfGlobal (G.1.1 and I.2.2).

```
BlockContext() {
  idx = (c < 2 ? c ^ 1 : 2) * 13 + s;
  idx *= (qf_thresholds.size() + 1);
  for (t : qf_thresholds) if (qf > t) idx++;
  for (i = 0; i < 3; i++) idx *= (lf_thresholds[i].size() + 1);
  lf_idx = 0;
  for (t : lf_thresholds[0]) if (qdc[0] > t) lf_idx++;
  lf_idx *= (lf_thresholds[2].size() + 1);
  for (t : lf_thresholds[2]) if (qdc[2] > t) lf_idx++;
  lf_idx *= (lf_thresholds[1].size() + 1);
  for (t : lf_thresholds[1]) if (qdc[1] > t) lf_idx++;
  return block_ctx_map[idx + lf_idx];
}
NonZerosContext(predicted) {
  if (predicted > 64) predicted = 64;
  if (predicted < 8) return BlockContext() + nb_block_ctx * predicted;
  return BlockContext() + nb_block_ctx * (4 + predicted Idiv 2);
}
CoeffFreqContext[64] = {
    0,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14,
    15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22,
    23, 23, 23, 23, 24, 24, 24, 24, 25, 25, 25, 25, 26, 26, 26, 26,
    27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30, 30};
CoeffNumNonzeroContext[64] = {
    0,     0,  31,  62,  62,  93,  93,  93,  93, 123, 123, 123, 123,
    152, 152, 152, 152, 152, 152, 152, 152, 180, 180, 180, 180, 180,
    180, 180, 180, 180, 180, 180, 180, 206, 206, 206, 206, 206, 206,
    206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206,
    206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206, 206};
CoefficientContext(k, non_zeros, num_blocks, size, prev) {
 non_zeros = (non_zeros + num_blocks - 1) Idiv num_blocks;
 k = k Idiv num_blocks;
 return (CoeffNumNonzeroContext[non_zeros] + CoeffFreqContext[k]) * 2 +
   prev + BlockContext() * 458 + 37 * nb_block_ctx;
}
```

The function `PredictedNonZeros(x, y)` is defined as follows (where `NonZeros` is defined below):

```
PredictedNonZeros(x, y) {
  if (x == 0 and y == 0) return 32;
  if (x == 0) return NonZeros(x, y - 1);
  if (y == 0) return NonZeros(x - 1, y);
  return (NonZeros(x, y-1) + NonZeros(x-1, y) + 1) >> 1;
}
```

After selecting the histogram and coefficient order, the decoder reads symbols from an entropy-coded stream, as specified in C.3.3. The decoder proceeds by decoding varblocks in raster order; for each varblock it reads channels Y, X, then B; if a channel is subsampled, its varblocks are skipped unless the varblock corresponds to the top-left corner of a non-subsampled varblock. For the purposes of this ordering, each varblock corresponds to its top-left block. For each varblock of size `W` × `H`, covering `num_blocks = (W / 8) * (H / 8)` blocks, the decoder reads an integer `non_zeros` using `DecodeHybridVarLenUint(NonZerosContext(PredictedNonZeros(x, y)) + offset)`. The decoder then sets the `NonZeros(x, y)` value for each block in the current varblock as follows: for each `i` in [0, `W / 8`) and `j` in [0, `H / 8`), `NonZeros(x + i, y + j)` is set to `(non_zeros + num_blocks - 1) Idiv num_blocks`. Let `size = W * H`. For `k` in the range [`num_blocks`, `size`), the decoder reads an integer `ucoeff` from the codestream, using `DecodeHybridVarLenUint(CoefficientContext(k, non_zeros, num_blocks, size, prev) + offset)`, where `prev` is computed as specified in the following code:

```
if (k == num_blocks) {
  if (non_zeros > size / 16) prev = 0;
  else prev = 1;
} else {
  if ([[decoded coefficient at position (k - 1) is 0]]) prev = 0;
  else prev = 1;
}
```

The decoder then sets the quantized HF coefficient in the position corresponding to index `order[p][s][c][k]` to `UnpackSigned(ucoeff)`, where `p` is the index of the current pass and `s` and `c` are the Order ID and current channel index as above. If `ucoeff != 0`, the decoder decreases `non_zeros` by 1. If `non_zeros` reaches 0, the decoder stops decoding further coefficients for the current block.

If this is not the first pass, the decoder adds decoded HF coefficients to previously-decoded ones.

I.5 Adaptive quantization

I.5.1 General

In var-DCT mode, quantized LF and HF coefficients qX, qY and qB are converted into values dX, dY and dB as specified in I.5.2 and I.5.3, respectively.

I.5.2 LF dequantization

If the **kUseLfFrame** flag in `frame_header` is set, this subclause is skipped for that frame. Let `mXDC`, `mYDC` and `mBDC` be the three per-channel LF dequantization multipliers (see G.1.2).

For each LF group, the dequantization process is influenced by the number `extra_precision`, as described in G.2.3. Dequantized values are computed from the quantized values `qX`, `qY`, `qB` in LfQuant (G.2.2) as specified by the following code:

```
dX = mXDC * qX / (1 << extra_precision);
dY = mYDC * qY / (1 << extra_precision);
dB = mBDC * qB / (1 << extra_precision);
```

After dequantizing LF coefficients and applying chroma from luma (I.6), the decoder applies the following adaptive smoothing algorithm, unless the **kSkipAdaptiveLFSmoothing** flag is set in `frame_header`. If this adaptive smoothing procedure is applied, no channel is subsampled.

For each LF sample of the image that is not in the first or last row or column, the decoder computes a weighted average `wa` of the 9 samples that are in neighbouring rows and columns, using a weight of 0.05226273532324128 for the current sample `s`, 0.20345139757231578 for horizontally/vertically adjacent samples, and 0.0334829185968739 for diagonally adjacent samples. Let `gap` = max(0.5, abs($wa_X - s_X$)/`mXDC`, abs($wa_Y - s_Y$)/`mYDC`, abs($wa_B - s_B$)/`mBDC`). The smoothed value is `(wa - s) * max(0, 3 - 4 * gap) + s`.

After applying this smoothing, the decoder uses the process described in I.8 to compute the top-left `X/8` × `Y/8` coefficients of each varblock of size X × Y, using the corresponding `X/8` × `Y/8` samples from the dequantized LF image.

I.5.3 HF dequantization

Every quantized HF coefficient quant is first bias-adjusted as specified by the following code, depending on its channel (0 for X, 1 for Y or 2 for B).

```
oim = metadata.opsin_inverse_matrix;
if (abs(quant) <= 1) quant *= oim.quant_bias[channel];
else quant -= oim.quant_bias_numerator / quant;
```

The resulting `quant` is then multiplied by a per-block multiplier `Mul` based on the value of `HfMul` at the coordinates of the 8 × 8 rectangle containing the current sample: `Mul = (1 << 16) / (quantizer.global_scale * HfMul)`. For the X and B channels, it is then multiplied by by `pow(0.8, frame_header.x_qm_scale - 2)` and `pow(0.8, frame_header.b_qm_scale - 2)`, respectively.

The final dequantized value is obtained by multiplying the result by a multiplier defined by the channel, the transform type and the coefficient index inside the varblock, as specified in I.2.4.

I.6 Chroma from luma

This clause is skipped if any channel is subsampled.

Each X, Y and B coefficient is reconstructed from the dequantized coefficients dX, dY and dB using a linear chroma from luma model. The reconstruction uses the colour correlation coefficient multipliers kX and kB (computed from constants defined in I.2.3) to restore the correlation between the X/B and the Y channel, as specified by the following code:

```
kX = base_correlation_x + x_factor / colour_factor;
kB = base_correlation_b + b_factor / colour_factor;
Y = dY;
X = dX + kX * Y;
B = dB + kB * Y;
```

For LF coefficients, `x_factor` and `b_factor` correspond to `x_factor_lf` − 128 and `b_factor_lf` − 128, respectively (I.2.3). LF coefficients are not modified if `flags`.**kUseLfFrame** is `true`. For HF coefficients, `x_factor` and `b_factor` are values from `XFromY` and `BFromY` (G.2.4), respectively, at the coordinates of the 64 × 64 rectangle containing the current sample.

I.7 Forward and inverse DCT

I.7.1 General

In **kVarDCT** mode, the decoder applies forward and inverse two-dimensional Discrete Cosine Transforms as specified in I.7.3, I.8, and I.9. In these clauses, a matrix of size RxC refers to a matrix with R rows and C columns.

I.7.2 One-dimensional DCT and IDCT

One-dimensional discrete cosine transforms and their inverses (DCT) are computed as follows. The input is a vector of size s, where s is a power of two.

Let *in* and *out* denote the inputs and outputs. Then the forward DCT transform is defined as:

$$out_k = \frac{1}{s}\left(k == 0 ? 1 : \sqrt{2}\right) \sum_{n=0}^{s-1} in_n \cdot \cos\left(\frac{\pi k}{s}\left(n + \frac{1}{2}\right)\right) \tag{I.1}$$

The inverse DCT transform (IDCT) is defined as follows:

$$in_k = out_0 + \sum_{n=1}^{s-1} \sqrt{2} \cdot out_n \cdot \cos\left(\frac{\pi n}{s}\left(k + \frac{1}{2}\right)\right) \tag{I.2}$$

I.7.3 Two-dimensional DCT and IDCT

In this subclause, Transpose is the matrix transpose and ColumnIDCT/ColumnDCT performs a one dimensional IDCT/DCT on each column vector of the given matrix as defined in I.7.2. The DCT transform of a RxC matrix DCT_2D(samples) is specified by the following code:

```
dct1 = ColumnDCT(samples);
dct1_t = Transpose(dct1);
dct2 = ColumnDCT(dct1_t);
if (C > R) result = Transpose(dct2);
else result = dct2;
```

NOTE The final Transpose ensures the DCT result has the same shape as the original varblock if the original varblock has more columns than rows, and has the shape of its transpose otherwise.

The inverse DCT (IDCT) to obtain a RxC matrix of samples IDCT_2D(coefficients) is specified by the following code:

```
if (C > R) dct2 = Transpose(coefficients);
else dct2 = coefficients;
dct1_t = ColumnIDCT(dct2);
dct1 = Transpose(dct1_t);
varblock = ColumnIDCT(dct1);
```

I.8 LLF coefficients from downsampled image

This subclause specifies how to obtain the low frequency DCT coefficients LLF from an 8 times downsampled image (LF), for each `DctSelect` value.

The input is a matrix with `bwidth / 8` columns and `bheight / 8` rows containing the LF coefficients, or equivalently, the downsampled image. The possible values for `bwidth` and `bheight` are 8, 16, 32, 64, 128 and 256.

The output is a matrix with `max(bwidth, bheight) / 8` columns and `min(bwidth, bheight) / 8` rows. If the caller requires a matrix of a larger size, the decoder zero-initializes the other elements. The following code specifies the function `ScaleF`.

```
ScaleF(c, b) {
    inverse_scale_f = cos(c * pi / (2 * b)) * cos(c * pi / b) * cos(2 * c * pi / b);
    return 1 / inverse_scale_f;
}
```

For `DctSelect` types DCT8×8, DCT8×16, DCT8×32, DCT16×8, DCT16×16, DCT32×8, DCT32×16, DCT32×32, DCT32×64, DCT64×32, DCT64×64, DCT64×128, DCT128×64, DCT128×128, DCT128×256, DCT256×128, and DCT256×256 the output is computed as specified by the following code:

```
cx = max(bwidth, bheight) / 8;
cy = min(bwidth, bheight) / 8;
dc = DCT_2D(input);
for (y = 0; y < cy; y++)
  for (x = 0; x < cx; x++)
    output(x, y) = dc(x, y) * ScaleF(y, cy) * ScaleF(x, cx);
```

For `DctSelect` types Hornuss, DCT2×2, DCT4×4, DCT8×4, DCT4×8, AFV0, AFV1, AFV2, AFV3, the output is equal to the input.

I.9 Coefficients to samples

I.9.1 General

Each of the subsequent subclauses in I.9 specifies how the decoder converts the coefficients to pixels where `DctSelect` matches one of the types listed in the heading of the subclause.

I.9.2 DCT*R*×*C* with *R*,*C* ≥ 8

The *R* × *C* matrix of samples are obtained using `samples = IDCT_2D(coefficients)` for a `DctSelect` value of DCT*R*×*C*.

I.9.3 DCT2×2

The samples of a DCT2×2 varblock are computed from an 8×8 block of coefficients. `AuxIDCT2x2(block, S)` is specified by the following code, in which `block` is a matrix of size S×S, and S is a power of two:

```
AuxIDCT2x2(block, S) {
  result = block;
  num_2x2 = S / 2;
  for (y = 0; y < num_2x2; y++)
    for (x = 0; x < num_2x2; x++) {
      c00 = block(x, y);
      c01 = block(num_2x2 + x, y);
      c10 = block(x, y + num_2x2);
      c11 = block(num_2x2 + x, y + num_2x2);
      r00 = c00 + c01 + c10 + c11;
      r01 = c00 + c01 - c10 - c11;
      r10 = c00 - c01 + c10 - c11;
      r11 = c00 - c01 - c10 + c11;
      result(x * 2, y * 2) = r00;
      result(x * 2 + 1, y * 2) = r01;
      result(x * 2, y * 2 + 1) = r10;
      result(x * 2 + 1, y * 2 + 1) = r11;
    }
  return result;
}
```

When block is larger than S×S, `AuxIDCT2x2` modifies only the top-left S×S cells. In that case, the decoder copies the rest of the cells from block to result. The inverse DCT2×2 is specified by the following code:

```
block = AuxIDCT2x2(block, 2);
block = AuxIDCT2x2(block, 4);
samples = AuxIDCT2x2(block, 8);
```

I.9.4 DCT4×4

The samples are computed from the 8×8 coefficient matrix as specified by the following code, in which `block` is a 4×4 matrix, and `sample` is the answer represented here as a 2×2 matrix whose elements are 4×4 matrices.

```
dcs = AuxIDCT2x2(coefficients, 2);
for (y = 0; y < 2; y++)
```

```
  for (x = 0; x < 2; x++) {
    for (iy = 0; iy < 4; iy++)
      for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
        block(ix, iy) = coefficients(x + ix * 2, y + iy * 2);
    block(0, 0) = dcs(x, y);
    sample(x, y) = IDCT_2D(block);
  }
```

The decoder re-shapes the sample array to an 8×8 matrix using `result(4 * i + k, 4 * j + L) = sample(i, j, k, L)`.

I.9.5 Hornuss

The 8×8 input coefficients are transformed to 8×8 samples as specified by the following code:

```
dcs = AuxIDCT2x2(coefficients, 2);
for (y = 0; y < 2; y++)
  for (x = 0; x < 2; x++) {
    block_lf = dcs(x, y);
    residual_sum = 0;
    for (iy = 0; iy < 4; iy++)
      for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
        residual_sum += coefficients(x + ix * 2, y + iy * 2);
    sample(4 * x + 1, 4 * y + 1) = block_lf - residual_sum / 16.0;
    for (iy = 0; iy < 4; iy++)
      for (ix = 0; ix < 4; ix++) {
        if (ix == 1 and iy == 1) continue;
        sample(x * 4 + ix, y * 4 + iy) =
            coefficients(x + ix * 2, y + iy * 2) +
            sample(4 * x + 1, 4 * y + 1);
      }
    sample(4 * x, 4 * y) =
        coefficients(x + 2, y + 2) + sample(4 * x + 1, 4 * y + 1);
  }
```

I.9.6 DCT8×4

The 8×8 samples are reconstructed from the 8×8 coefficients by dividing them into two 8×4 vertical blocks `samples_8x4`, as specified by the following code, in which `coeffs_4x8` denotes a temporary 4 × 8 matrix.

```
coef0 = coefficients(0, 0); coef1 = coefficients(0, 1);
dcs = {coef0 + coef1, coef0 - coef1};
for (x = 0; x < 2; x++) {
  coeffs_8x4(0, 0) = dcs[x];
  for (iy = 0; iy < 4; iy++)
    for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++) {
      coeffs_4x8(ix, iy) = coefficients(ix, x + iy * 2);
    }
  samples_8x4[x] = IDCT_2D(coeffs_4x8);
}
```

I.9.7 DCT4×8

The 8×8 samples are reconstructed from the 8×8 coefficients by dividing them into two 4×8 horizontal blocks `samples_4x8`, as specified by the following code, in which `coeffs_4x8` denotes a temporary 4 × 8 matrix.

```
coef0 = coefficients(0, 0); coef1 = coefficients(0, 1);
dcs = {coef0 + coef1, coef0 - coef1};
for (y = 0; y < 2; y++) {
  coeffs_4x8(0, 0) = dcs[y];
  for (iy = 0; iy < 4; iy++)
    for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++)
      coeffs_4x8(ix, iy) = coefficients(ix, y + iy * 2);
  samples_4x8[y] = IDCT_2D(coeffs_4x8);
}
```

I.9.8 AFV0, AFV1, AFV2, AFV3

The 8×8 samples are reconstructed from the 8×8 coefficients by dividing them into two 4×4 square blocks and one horizontal 4×8, as specified by the following code. Four temporary 4×4 matrices `coeffs_` afv, samples_afv, coeffs_4x4 and samples_4x4, as well as two temporary 4×8 matrices coeffs_4x8, and samples_4x8 are used. The values of flip_x and flip_y are defined as follows: for AFV*n*, flip_x is equal to *n* & 1 and flip_y is equal to *n* Idiv 2.

```
coeffs_afv[0] = (coefficients(0, 0) + coefficients(0, 1) + coefficients(1, 0)) * 4.0;
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
    coeff_afv[iy * 4 + ix] = coefficients(ix * 2, iy * 2);
for (iy = 0; iy < 4; iy++) {
  for (ix = 0; ix < 4; ix++) {
    sample = 0;
    for (j = 0; j < 16; ++j) sample += coeff_afv[j] * AFVBasis[j][iy * 4 + ix];
    samples_afv(ix, iy) = sample;
  }
}
for (iy = 0; iy < 4; iy++)
  for (ix = 0; ix < 4; ix++)
    samples(flip_x * 4 + ix, flip_y * 4  + iy) =
      samples_afv(flip_x == 1 ? 3 - ix : ix, flip_y == 1 ? 3 - iy : iy);
coeffs_4x4(0, 0) = coefficients(0, 0) - coefficients(1, 0) + coefficients(0, 1);
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 4; ix++)
    coeffs_4x4(ix, iy) = coefficients(ix * 2 + 1, iy * 2);
samples_4x4 = IDCT_2D(coeffs_4x4);
for (iy = 0; iy < 4; iy++)
  for (ix = 0; ix < 4; ix++)
    samples((flip_x == 1 ? 0 : 4) + ix, flip_y * 4  + iy) =
      samples_4x4(ix, iy);
coeffs_4x8(0, 0) = coefficients(0, 0) - coefficients(0, 1);
for (iy = 0; iy < 4; iy++)
  for (ix = (iy == 0 ? 1 : 0); ix < 8; ix++)
    coeffs_4x8(ix, iy) = coefficients(ix, 1 + iy * 2);
samples_4x8 = IDCT_2D(coeffs_4x8);
for (iy = 0; iy < 4; iy++)
  for (ix = 0; ix < 8; ix++)
    samples(ix, (flip_y == 1 ? 0 : 4) + iy) = samples_4x8(ix, iy);
```

The AFV transform uses the following orthonormal basis:

AFVBasis[16][16] = { {0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25, 0.25}, {0.876902929799142, 0.2206518106944235, −0.10140050393753763, −0.1014005039375375, 0.2206518106944236, −0.10140050393753777, −0.10140050393753772, −0.10140050393753763, −0.10140050393753758, −0.10140050393753769, −0.1014005039375375, −0.10140050393753768, −0.10140050393753768, −0.10140050393753759, −0.10140050393753763, −0.10140050393753741}, {0.0, 0.0, 0.40670075830260755, 0.44444816619734445, 0.0, 0.0, 0.19574399372042936, 0.2929100136981264, −0.40670075830260716, −0.19574399372042872, 0.0, 0.11379074460448091, −0.44444816619734384, −0.29291001369812636, −0.1137907446044814, 0.0}, {0.0, 0.0, −0.21255748058288748, 0.3085497062849767, 0.0, 0.4706702258572536, −0.1621205195722993, 0.0, −0.21255748058287047, −0.16212051957228327, −0.47067022585725277, −0.1464291867126764, 0.3085497062849487, 0.0, −0.14642918671266536, 0.4251149611657548,}, {0.0, −0.7071067811865474, 0.0, 0.0, 0.7071067811865476, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}, {−0.4105377591765233, 0.6235485373547691, −0.06435071657946274, −0.06435071657946266, 0.6235485373547694, −0.06435071657946284, −0.0643507165794628, −0.06435071657946274, −0.06435071657946272, −0.06435071657946279, −0.06435071657946266, −0.06435071657946277, −0.06435071657946277, −0.06435071657946273, −0.06435071657946274, −0.0643507165794626}, {0.0, 0.0, −0.4517556589999482, 0.15854503551840063, 0.0, −0.04038515160822202, 0.0074182263792423875, 0.39351034269210167, −0.45175565899994635, 0.007418226379244351, 0.1107416575309343, 0.08298163094882051, 0.15854503551839705, 0.3935103426921022, 0.0829816309488214, −0.45175565899994796}, {0.0, 0.0, −0.304684750724869, 0.5112616136591823, 0.0, 0.0, −0.290480129728998, −0.06578701549142804, 0.304684750724884, 0.2904801297290076, 0.0, −0.23889773523344604, −0.5112616136592012, 0.06578701549142545, 0.23889773523345467, 0.0}, {0.0, 0.0, 0.3017929516615495, 0.25792362796341184, 0.0, 0.16272340142866204, 0.09520022653475037, 0.0, 0.3017929516615503, 0.09520022653475055, −0.16272340142866173, −0.35312385449816297, 0.25792362796341295, 0.0, −0.3531238544981624, −0.6035859033230976,}, {0.0, 0.0, 0.40824829046386274, 0.0, 0.0, 0.0, 0.0, −0.4082482904638628, −0.4082482904638635, 0.0, 0.0, −0.40824829046386296, 0.0, 0.4082482904638634, 0.408248290463863, 0.0}, {0.0, 0.0, 0.1747866975480809, 0.0812611176717539, 0.0, 0.0, −0.3675398009862027, −0.307882213957909, −0.17478669754808135, 0.3675398009862011, 0.0, 0.4826689115059883, −0.08126111767175039, 0.30788221395790305, −0.48266891150598584, 0.0}, {0.0, 0.0, −0.21105601049335784, 0.18567180916109802, 0.0, 0.0, 0.49215859013738733, −0.38525013709251915, 0.21105601049335806, −0.49215859013738905, 0.0, 0.17419412659916217, −0.18567180916109904, 0.3852501370925211, −0.1741941265991621, 0.0}, {0.0, 0.0, −0.14266084808807264, −0.3416446842253372, 0.0, 0.7367497537172237, 0.24627107722075148, −0.08574019035519306, −0.14266084808807344, 0.24627107722075137, 0.14883399227113567, −0.04768680350229251, −0.3416446842253373, −0.08574019035519267, −0.047686803502292804, −0.14266084808807242}, {0.0, 0.0, −0.13813540350758585, 0.3302282550303788, 0.0, 0.08755115000587084, −0.07946706605909573, −0.4613374887461511, −0.13813540350758294, −0.07946706605910261, 0.49724647109535086, 0.12538059448563663, 0.3302282550303805, −0.4613374887461554, 0.12538059448564315, −0.13813540350758452}, {0.0, 0.0, −0.17437602599651067, 0.0702790691196284, 0.0, −0.2921026642334881, 0.3623817333531167, 0.0, −0.1743760259965108, 0.36238173335311646, 0.29210266423348785, −0.4326608024727445, 0.07027906911962818, 0.0, −0.4326608024727457, 0.34875205199302267,}, {0.0, 0.0, 0.11354987314994337, −0.07417504595810355, 0.0, 0.19402893032594343, −0.435190496523228, 0.21918684838857466, 0.11354987314994257, −0.4351904965232251, 0.5550443808910661, −0.25468277124066463, −0.07417504595810233, 0.2191868483885728, −0.25468277124066413, 0.1135498731499429} }.

Annex J
(normative)

Restoration filters

J.1 General

The decoder supports two kinds of restoration filters: a Gabor-like transform and an edge-preserving filter.

The `frame_header` (see F.2) contains a `restoration_filter` bundle, which is defined in Table J.1.

Table J.1 — RestorationFilter bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	Bool()	`true`	`gab`
`gab`	Bool()	`false`	`gab_custom`
`gab_custom`	F16()	0.115169525	`gab_x_weight1`
`gab_custom`	F16()	0.061248592	`gab_x_weight2`
`gab_custom`	F16()	0.115169525	`gab_y_weight1`
`gab_custom`	F16()	0.061248592	`gab_y_weight2`
`gab_custom`	F16()	0.115169525	`gab_b_weight1`
`gab_custom`	F16()	0.061248592	`gab_b_weight2`
`!all_default`	u(2)	2	`epf_iters`
`!all_default and epf_iters and encoding ==` **kVarDCT**	Bool()	`false`	`epf_sharp_custom`
`epf_sharp_custom`	F16()	{0, 1/7, 2/7, 3/7, 4/7, 5/7, 6/7, 1}	`epf_sharp_lut[8]`
`!all_default and epf_iters`	Bool()	`false`	`epf_weight_custom`
`epf_weight_custom`	F16()	{40.0, 5.0, 3.5}	`epf_channel_scale[[3]]`
`epf_weight_custom`	u(32)	0	(ignored)
`!all_default and epf_iters`	Bool()	`false`	`epf_sigma_custom`
`epf_sigma_custom and encoding ==` **kVarDCT**	F16()	0.46	`epf_quant_mul`
`epf_sigma_custom`	F16()	0.9	`epf_pass0_sigma_scale`
`epf_sigma_custom`	F16()	6.5	`epf_pass2_sigma_scale`
`epf_sigma_custom`	F16()	2 / 3	`epf_border_sad_mul`
`!all_default and epf_iters and encoding ==` **kModular**	F16()	1.0	`epf_sigma_for_modular`
`!all_default`	Extensions		`extensions`

`gab` and `epf_iters` determine whether the Gabor-like transform (J.3) and the edge-preserving filter (J.4) are applied. The other fields parameterize any enabled filter(s).

If `restoration_filter.gab`, a convolution is applied to the entire frame as defined in J.3.

If `restoration_filter.epf_iters`, an edge-preserving adaptive filter is applied to the entire frame immediately after applying the Gabor-like transform, as defined in J.4.

In the case where `frame_header.do_YCbCr`, any subsampled channels (according to `frame_header.jpeg_upsampling` are first upsampled as specified in J.2, before any restoration filter is applied.

J.2 Simple upsampling

Horizontal (vertical) upsampling is defined as follows: the channel width (height) is multiplied by two (but limited to the image dimensions), and every subsampled pixel `B` with left (top) neighbour `A` and right (bottom) neighbour `C` is replaced by two upsampled pixels with values `0.25 * A + 0.75 * B` and `0.75 * B + 0.25 * C`. At the image edges, the missing neighbors are replicated from the edge pixel, i.e. the mirroring procedure of 5.2 is used.

NOTE This upsampling procedure corresponds to the so-called "triangle filter" that is commonly implemented in existing JPEG image decoders.

J.3 Gabor-like transform

In this subclause, C denotes x, y, or b according to the channel currently being processed. The decoder applies a convolution to each channel of the entire image with the following symmetric 3×3 kernel: the unnormalized weight for the center is 1, its four neighbours (top, bottom, left, right) are `restoration_filter.gab_C_weight1` and the four corners (top-left, top-right, bottom-left, bottom-right) are `restoration_filter.gab_C_weight2`. These weights are rescaled uniformly before convolution, such that the nine kernel weights sum to 1.

When the convolution references input pixels with coordinates `cx`, `cy` outside the bounds of the original image, the decoder instead accesses the pixel at coordinates `Mirror(cx, cy)` (5.2).

J.4 Edge-preserving filter

J.4.1 General

In this subclause, `rf` denotes `restoration_filter`.

The filter is only performed if `rf.epf_iters > 0`. If it is performed, the filter operates in up to three steps. The output of each step is used as an input for the following step.

In each step, an L1 distance metric is defined as specified in J.4.2. It is used to compute weights for each reference pixel (J.4.3), and then the pixel is updated using a weighted average (J.4.4).

The first step is only done if `rf.epf_iters == 3`. In this step, for each reference pixel, the filter outputs a pixel which is a weighted sum of the reference pixel and each of the twelve neighbouring pixels that have a L1 distance of at most 2.

The second step is always done (if `rf.epf_iters > 0`). In this step, for each reference pixel, the filter outputs a pixel which is a weighted sum of the reference pixel and each of the four neighbour pixels (top, bottom, left and right of the current one).

In both the first and the second step, the distance for each weight is computed over two cross shapes consisting of five pixels (center, top, bottom, left and right), where the centers are the reference pixel and the pixel corresponding to the weight.

The third step is only done if `rf.epf_iters >= 2`. In this step, for each reference pixel, the filter outputs a pixel which is a weighted sum of the reference pixel and each of the four neighbour pixels (top, bottom, left and right of the current one). The distance for each weight is computed based on the reference pixel and the pixel corresponding to the weight.

In this subclause, the filter may reference guide or input pixels with coordinates `cx`, `cy` outside the valid bounds. The decoder behaves as if every such access were redirected to coordinates `Mirror(cx, cy)` (5.2).

J.4.2 Distances

For a reference pixel in position (x, y), the decoder computes the distance value for the given neighbouring pixel in position (x+cx, y+cy) as specified by the following code, in which sample(x, y, c) is the sample in channel c of the image in coordinates (x, y).

```
DistanceStep0and1(x, y, cx, cy) {
  dist = 0; coords = { {0, 0}, {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
  for (c = 0; c < 3; c++) {
    for (/* (ix, iy) in coords */) {
      dist += abs(sample(x + ix, y + iy, c) -
        sample(x + cx + ix, y + cy + iy, c)) * rf.epf_channel_scale[c];
    }
  }
  return dist;
}
DistanceStep2(x, y, cx, cy) {
  dist = 0;
  for (c = 0; c < 3; c++) {
    dist += abs(sample(x, y, c) -
      sample(x + cx, y + cy, c)) * rf.epf_channel_scale[c];
  }
  return dist;
}
```

J.4.3 Weights

The convolution kernel for each reference pixel is determined by the five weights of its neighbours and of the reference pixel, computed using a decreasing function Weight() specified by the following code:

```
Weight(distance, sigma) {
  step = /* 0 if first step, 1 if second step, 2 if third step */;
  step_multiplier = 1.65 * {rf.epf_pass0_sigma_scale, 1, rf.epf_pass2_sigma_scale};
  if (/* either coordinate of the reference sample is 0 or 7 UMod 8 */)
       position_multiplier = rf.epf_border_sad_mul;
  else position_multiplier = 1;
  inv_sigma = step_multiplier[step] * 4 * (1 - sqrt(0.5)) / sigma;
  scaled_distance = position_multiplier * distance;
  v = 1 - scaled_distance * inv_sigma;
  if (v <= 0) return 0;
  return v;
}
```

Let quantization_width and sharpness denote the values of Mul (as defined in I.5.3) and Sharpness (G.2.4) at the coordinates of the 8 × 8 rectangle containing the reference pixel.

sigma is then computed as specified by the following code if the frame encoding is **kVarDCT**, else it is set to rf.epf_sigma_for_modular.

```
sigma = quantization_width * rf.epf_quant_mul;
sigma *= rf.epf_sharp_lut[sharpness];
```

If sigma < 0.3 for a given varblock, the decoder skips all steps on the pixels of that block: for each step, the output samples are the same as the input sample in the given block.

J.4.4 Weighted average

The output of each step of the filter is computed as specified by the following code, in which sigma is the sigma value for the block containing sample (x, y), and input(x, y, c) is the input sample in channel c and position (x, y).

```
if (/* step 0 */) {
  kernel_coords = { {0, 0}, {-1, 0}, {1, 0}, {0, -1}, {0, 1},
                    {1, -1}, {1, 1}, {-1, 1}, {-1, -1}, {-2, 0},
                    {2, 0}, {0, 2}, {0, -2} };
} else {
  kernel_coords = { {0, 0}, {-1, 0}, {1, 0}, {0, -1}, {0, 1} };
}
sum_weights = 0;
```

```
sum_channels = {0, 0, 0};
for (/* (ix, iy) in kernel_coords */) {
   if (/* step 0 or 1 */) {
     distance = DistanceStep0and1(x, y, ix, iy);
   } else {
     distance = DistanceStep2(x, y, ix, iy);
   }
   weight = Weight(distance, sigma);
   sum_weights += weight;
   for (c = 0; c < 3; c++) {
     sum_channels[c] += input(x + ix, y + iy, c) * weight;
   }
}
for (c = 0; c < 3; c++) {
  output(x, y, c) = sum_channels[c] / sum_weights;
}
```

Annex K
(normative)

Image features

K.1 General

Each frame may include alternate representations of image contents (splines, patches and noise). If any of them are enabled (as indicated by `frame_header.flags`), the decoder draws them on top of the frame as specified in subclauses K.3 to K.5.

In case `frame_header.upsampling > 1` and/or `max(frame_header.ec_upsampling) > 1`, the corresponding colour and/or extra channels are first upsampled as specified in K.2.

K.2 Non-separable upsampling

Horizontal and vertical upsampling are always performed simultaneously. There are three ways of doing upsampling: 2x, 4x and 8x. Consider k to be 2, 4 or 8. From each sample p, $k \times k$ new samples are obtained when upscaling is performed. Each new sample in such a $k \times k$ block is computed as a linear combination of a 5 × 5 window W of samples in the original image centered around p, where boundary samples are mirrored as in 5.2, and the resulting sample is clamped to the range $[a, b]$ where a and b are the minimum and maximum of the samples in W.

The coefficients of this linear combination depend only on the new position in the $k \times k$ block and are viewed as a 5 × 5 matrix. Hence in total 25 k^2 weights are used.

When doing $k \times k$ upsampling, for a new sample at position (`kx`,`ky`) within the $k \times k$ block, the weight of the original sample at position (`ix`,`iy`) in the 5 × 5 window W is computed as follows:

```
j = (ky < k/2) ? (iy + 5 * ky) : ((4 - iy) + 5 * (k - 1 - ky));
i = (kx < k/2) ? (ix + 5 * kx) : ((4 - ix) + 5 * (k - 1 - kx));
y = min(i, j); x = max(i, j);
index = 5 * k * y / 2 - y * (y - 1) / 2 + x - y;
```

The weight of the sample is then `metadata.upk_weight[index]` (D.3).

EXAMPLE For k = 2, the 2 × 2 matrices of 5 × 5 weights are as follows, where the numbers denote indices in the array `metadata.up2_weight[15]`:

0	1	2	3	4		4	3	2	1	0
1	5	6	7	8		8	7	6	5	1
2	6	9	10	11		11	10	9	6	2
3	7	10	12	13		13	12	10	7	3
4	8	11	13	14		14	13	11	8	4
4	8	11	13	14		14	13	11	8	4
3	7	10	12	13		13	12	10	7	3
2	6	9	10	11		11	10	9	6	2
1	5	6	7	8		8	7	6	5	1
0	1	2	3	4		4	3	2	1	0

The default values `d_up2`, `d_up4` and `d_up8` of the arrays `metadata.up2_weight[15]`, `metadata.up4_weight[55]` and `metadata.up8_weight[210]` are defined as follows:

`d_up2` = { −0.01716200, −0.03452303, −0.04022174, −0.02921014, −0.00624645, 0.14111091, 0.28896755, 0.00278718, −0.01610267, 0.56661550, 0.03777607, −0.01986694, −0.03144731, −0.01185068, −0.00213539 }.

`d_up4` = { −0.02419067, −0.03491987, −0.03693351, −0.03094285, −0.00529785, −0.01663432, −0.03556863, −0.03888905, −0.03516850, −0.00989469, 0.23651958, 0.33392945, −0.01073543, −0.01313181, −0.03556694, 0.13048175, 0.40103025, 0.03951150, −0.02077584, 0.46914198, −0.00209270, −0.01484589, −0.04064806, 0.18942530, 0.56279892, 0.06674400, −0.02335494, −0.03551682, −0.00754830, −0.02267919, −0.02363578, 0.00315804, −0.03399098, −0.01359519, −0.00091653, −0.00335467, −0.01163294, −0.01610294, −0.00974088, −0.00191622, −0.01095446, −0.03198464, −0.04455121, −0.02799790, −0.00645912, 0.06390599, 0.22963888, 0.00630981, −0.01897349, 0.67537268, 0.08483369, −0.02534994, −0.02205197, −0.01667999, −0.00384443 }.

`d_up8` = { −0.02928613, −0.03706353, −0.03783812, −0.03324558, −0.00447632, −0.02519406, −0.03752601, −0.03901508, −0.03663285, −0.00646649, −0.02066407, −0.03838633, −0.04002101, −0.03900035, −0.00901973, −0.01626393, −0.03954148, −0.04046620, −0.03979621, −0.01224485, 0.29895328, 0.35757708, −0.02447552, −0.01081748, −0.04314594, 0.23903219, 0.41119301, −0.00573046, −0.01450239, −0.04246845, 0.17567618, 0.45220643, 0.02287757, −0.01936783, −0.03583255, 0.11572472, 0.47416733, 0.06284440, −0.02685066, 0.42720050, −0.02248939, −0.01155273, −0.04562755, 0.28689496, 0.49093869, −0.00007891, −0.01545926, −0.04562659, 0.21238920, 0.53980934, 0.03369474, −0.02070211, −0.03866988, 0.14229550, 0.56593398, 0.08045181, −0.02888298, −0.03680918, −0.00542229, −0.02920477, −0.02788574, −0.02118180, −0.03942402, −0.00775547, −0.02433614, −0.03193943, −0.02030828, −0.04044014, −0.01074016, −0.01930822, −0.03620399, −0.01974125, −0.03919545, −0.01456093, −0.00045072, −0.00360110, −0.01020207, −0.01231907, −0.00638988, −0.00071592, −0.00279122, −0.00957115, −0.01288327, −0.00730937, −0.00107783, −0.00210156, −0.00890705, −0.01317668, −0.00813895, −0.00153491, −0.02128481, −0.04173044, −0.04831487, −0.03293190, −0.00525260, −0.01720322, −0.04052736, −0.05045706, −0.03607317, −0.00738030, −0.01341764, −0.03965629, −0.05151616, −0.03814886, −0.01005819, 0.18968273, 0.33063684, −0.01300105, −0.01372950, −0.04017465, 0.13727832, 0.36402234, 0.01027890, −0.01832107, −0.03365072, 0.08734506, 0.38194295, 0.04338228, −0.02525993, 0.56408126, 0.00458352, −0.01648227, −0.04887868, 0.24585519, 0.62026135, 0.04314807, −0.02213737, −0.04158014, 0.16637289, 0.65027023, 0.09621636, −0.03101388, −0.04082742, −0.00904519, −0.02790922, −0.02117818, 0.00798662, −0.03995711, −0.01243427, −0.02231705, −0.02946266, 0.00992055, −0.03600283, −0.01684920, −0.00111684, −0.00411204, −0.01297130, −0.01723725, −0.01022545, −0.00165306, −0.00313110, −0.01218016, −0.01763266, −0.01125620, −0.00231663, −0.01374149, −0.03797620, −0.05142937, −0.03117307, −0.00581914, −0.01064003, −0.03608089, −0.05272168, −0.03375670, −0.00795586, 0.09628104, 0.27129991, −0.00353779, −0.01734151, −0.03153981, 0.05686230, 0.28500998, 0.02230594, −0.02374955, 0.68214326, 0.05018048, −0.02320852, −0.04383616, 0.18459474, 0.71517975, 0.10805613, −0.03263677, −0.03637639, −0.01394373, −0.02511203, −0.01728636, 0.05407331, −0.02867568, −0.01893131, −0.00240854, −0.00446511, −0.01636187, −0.02377053, −0.01522848, −0.00333334, −0.00819975, −0.02964169, −0.04499287, −0.02745350, −0.00612408, 0.02727416, 0.19446600, 0.00159832, −0.02232473, 0.74982506, 0.11452620, −0.03348048, −0.01605681, −0.02070339, −0.00458223}.

K.3 Patches

K.3.1 Patches decoding

If the **kPatches** flag in `frame_header` is not set, this subclause is skipped for that frame.

Otherwise, the patch dictionary contains a set of small image patches, to be added to the decoded image in specified locations as specified in K.3.

The decoder reads a set of 10 pre-clustered distributions D, according to C.1, and then reads the following values from a single stream as specified in C.3.3.

Each patch type has the following attributes:

— `width`, `height`: dimensions of the image patch.

— `ref`: index of the reference frame the patch is taken from

— `x0`, `y0`: coordinates of the top-left corner of the patch in the reference image

— `sample(x, y, c)`: the sample from channel `c` at the position (`x`, `y`) within the patch, corresponding to the sample from channel `c` at the position (`x0 + x`, `y0 + y`) in Reference[`ref`]. All referenced samples are within the bounds of the reference frame.

— `count`: the number of distinct positions

— `blending`: arrays of `count` blend mode information structures, which consists of arrays of `mode`, `alpha_channel` and `clamp`

— `x`, `y`: arrays of `count` positions where this patch is added to the image

The decoder first sets `num_patches = DecodeHybridVarLenUint(0)`. For i in the range [0, `num_patches`) in ascending order, the decoder decodes `patch[i]` as specified by the following code:

```
patch[i].ref = DecodeHybridVarLenUint(1);
patch[i].x0 = DecodeHybridVarLenUint(3);
patch[i].y0 = DecodeHybridVarLenUint(3);
patch[i].width = DecodeHybridVarLenUint(2) + 1;
patch[i].height = DecodeHybridVarLenUint(2) + 1;
patch[i].count = DecodeHybridVarLenUint(7) + 1;
for (j = 0; j < patch[i].count; j++) {
 if (j == 0) {
   patch[i].x[j] = DecodeHybridVarLenUint(4);
   patch[i].y[j] = DecodeHybridVarLenUint(4);
 } else {
   patch[i].x[j] = UnpackSigned(DecodeHybridVarLenUint(6)) + patch[i].x[j - 1];
   patch[i].y[j] = UnpackSigned(DecodeHybridVarLenUint(6)) + patch[i].y[j - 1];
 }
 /* the width x height rectangle with top-left coordinates (x, y)
    is fully contained within the frame */
 for (k = 0; k < num_extra + 1; k++) {
   mode = DecodeHybridVarLenUint(5);
   /* mode < 8 */
   patch[i].blending[j].mode[k] = mode;
   if ((mode > 3 and /*  there is more than 1 alpha channel */) {
     patch[i].blending[j].alpha_channel[k] = DecodeHybridVarLenUint(8);
     /* this is a valid index of an extra channel */
   }
   if (mode > 2) patch[i].blending[j].clamp[k] = DecodeHybridVarLenUint(9);
 }
}
```

K.3.2 Patches rendering

If the **kPatches** flag in `frame_header` is not set, this subclause is skipped for that frame.

After applying restoration filters (Annex J), for `i` in the range [0, `num_patches`), for `j` in the range [0, `patch[i].count`), and for `c` in the range [0, `num_extra`], the decoder blends `new_sample`, which is `patch[i].sample(ix, iy, d)` (from K.3.1), on top of `old_sample`, which is the sample in channel `d` of the current frame at position (`patch[i].x[j] + ix`, `patch[i].y[j] + iy`), to obtain a new sample `sample` which overwrites `old_sample`. Here `d` iterates over the three colour channels if `c == 0` and refers to the extra channel with index `c-1` otherwise, and `ix`, `iy` are iterated between zero and (`patch[i].width`, `patch[i].height`) respectively. The sample values `new_sample` are in the colour space *before* the inverse colour transforms from L.2, L.3 and L.4 are applied, but after the upsampling from J.2 and K.2.

The blending is performed according to `patch[i].blending[j].mode[c]`, as specified in Table K.1, where "the alpha channel" is the extra channel with index `k = patch[i].blending[j].alpha_channel[c]`. If `patch[i].blending[j].clamp[c]` is `true`, alpha values are clamped to the interval [0, 1] before blending.

Table K.1 — PatchBlendMode

name	value	meaning
kNone	0	Nothing is done: `sample = old_sample`
kReplace	1	Same as **kReplace** in Table F.7: `sample = new_sample`
kAdd	2	Same as **kAdd** in Table F.7: `sample = old_sample + new_sample`
kMul	3	Same as **kMul** in Table F.7: `sample = old_sample * new_sample`
kBlendAbove	4	Same as **kBlend** in Table F.7 (`new_sample` gets alpha-blended over `old_sample`)
kBlendBelow	5	Same as **kBlend** in Table F.7, except the roles of `new_sample` and `old_sample` are swapped (alpha-blend *under*)
kMulAddAbove	6	Same as **kMulAdd** in Table F.7
kMulAddBelow	7	Same as **kMulAdd** in Table F.7, except the roles of `new_sample` and `old_sample` are swapped

If the blending involves an alpha channel (`patch[i].blending[j].mode[c] > 3`), and if the colour channels are upsampled (`frame_header.upsampling > 1`), then the alpha channel has the same total upsampling factor as the colour channels: `frame_header.ec_upsampling[k] << ec_info[k].dim_shift == frame_header.upsampling`.

K.4 Splines

K.4.1 Splines decoding

Spline data is present in the codestream if and only if the **kSplines** flag in `frame_header` (F.2) is set. If so, it consists of an ANS stream (C.1). The decoder first reads six pre-clustered distributions as described in C.1, and then reads coordinates as specified by the following code:

```
manhattan_distance = 0;
num_splines = DecodeHybridVarLenUint(2) + 1;
last_x = 0; last_y = 0;
for (i = 0; i < num_splines; i++) {
  x = DecodeHybridVarLenUint(1); y = DecodeHybridVarLenUint(1);
  if (i != 0) {
    x = UnpackSigned(x) + last_x; y = UnpackSigned(y) + last_y;
  }
  sp_x[i] = x; sp_y[i] = y;
  last_x = x; last_y = y;
}
quant_adjust = UnpackSigned(DecodeHybridVarLenUint(0));
```

For each of `num_splines` splines, the following steps are applied to decode the control points and quantized coefficients. The number of control points (including the starting point) is read as `num_control_points = 1 + DecodeHybridVarLenUint(3)`. The result of applying double delta encoding to the control point coordinates (separately for x and y) and then dropping the very first control point is read as interleaved values (x1, y1, x2, y2, ...) via `UnpackSigned(DecodeHybridVarLenUint(4))`. That is, for a starting point (`sp_x`, `sp_y`), `DecodeDoubleDelta(sp_x, {x1, x2, …})` and `DecodeDoubleDelta(sp_y, {y1, y2, …})` give the x and y coordinates of the spline's control points, as specified in the following code:

```
DecodeDoubleDelta(starting_value, delta[n]) {
  /* Append starting_value to the list of decoded values */;
  current_value = starting_value;
  current_delta = 0;
  for (i = 0; i < n; ++i) {
    current_delta += delta[i];
    current_value += current_delta;
    manhattan_distance += abs(current_delta);
    /* Append current_value to the list of decoded values */;
  }
}
```

Two consecutive control points do not have identical (x,y) coordinates. Then, `UnpackSigned(DecodeHybridVarLenUint(5))` is used four times thirty-two times to obtain, in this order, sequences of thirty-two integers representing the following coefficients of the spline along its arc length:

— `q_dctX[32]`, the decorrelated and quantized DCT32 coefficients of the X channel of the spline;

— `q_dctY[32]`, the quantized DCT32 coefficients of the Y channel of the spline;

— `q_dctB[32]`, the decorrelated and quantized DCT32 coefficients of the B channel of the spline;

— `q_dctSigma[32]`, the quantized DCT32 coefficients of the σ parameter of the spline (defining its thickness).

These coefficients are dequantized and recorrelated (see I.2.3) as follows:

```
width_estimate = 0;
qa = quant_adjust >= 0 ? 1 + quant_adjust / 8 : 1 / (1 - quant_adjust / 8);
colorX = colorY = colorB = 0;
for (i=0; i < 32; i++) {
  colorX += ceil(abs(q_dctX[i]) / qa);
  colorY += ceil(abs(q_dctY[i]) / qa);
  colorB += ceil(abs(q_dctB[i]) / qa);
}
colorX += ceil(abs(base_correlation_x)) * colorY;
colorB += ceil(abs(base_correlation_b)) * colorY;
logcolor = max(1, ceil(log2(1 + max(colorX, colorY, colorB))));
for (i=0; i < 32; i++) {
  dctY[i] = q_dctY[i] * 0.075 / qa;
  dctX[i] = q_dctX[i] * 0.0042 / qa + base_correlation_x * dctY[i];
  dctB[i] = q_dctB[i] * 0.07 / qa + base_correlation_b * dctY[i];
  dctSigma[i] = q_dctSigma[i] * 0.3333 / qa;
  weight = max(1, ceil(abs(q_dctSigma[i]) / qa));
  width_estimate += weight * weight * logcolor;
}
estimated_area_reached = width_estimate * manhattan_distance;
```

NOTE The variable `estimated_area_reached` (and the auxiliary variables `width_estimate`, `manhattan_distance`, `colorX`, `colorY`, `colorB`, `weight`, and `logcolor`) involved in its computation) is not needed to decode the image; it is only needed to check conformance with the levels described in Annex M. This variable (including the divisions by `qa`) can be computed using only integer arithmetic and before the actual spline rendering is performed.

K.4.2 Splines rendering

If the **kSplines** flag in `frame_header` is not set, this subclause is skipped for that frame. Otherwise, all (centripetal Catmull-Rom) splines are rendered after patches (K.3), via pixel-by-pixel addition, as follows.

In this subclause, the following definitions apply:

```
Mirror(center, point) = center + (center - point);
ContinuousIDCT(dct, t) = dct[0] + /* sum for k = [1, 32):
                         sqrt(2) * dct[k] * cos(k * (pi / 32) * (t + 0.5)) */;
```

For each spline, given the decoded control point positions S and the dequantized and recorrelated DCT32 coefficients of the X, Y, B and σ values along the arc length (K.4.1), the decoder renders the spline as specified by the following code. The vectors like S are vectors of points, which are pairs of (x,y) coordinates for which arithmetic operations are defined element-wise, e.g., in the Mirror() function defined above, the result of `Mirror((x,y), (a,b))` is the point `(x+x-a, y+y-b)`.

```
if (S.size() == 1) {
  upsampled_spline.push_back(S[0]);
} else {
  extended.push_back(Mirror(S[0], S[1]));
  extended.append(S);
  extended.push_back(Mirror(S[S.count - 1], S[S.count - 2]));
  for (i = 0; i < extended.size() - 3; ++i) {
    for (k = 0; k < 4; ++k) { p[k] = extended[i + k]; }
    upsampled_spline.push_back(p[1]);
    t[0] = 0;
    for (k = 1; k < 4; ++k) t[k] = t[k - 1] +
        pow(pow(p[k].x - p[k - 1].x, 2) + pow(p[k].y - p[k - 1].y, 2), 0.25);

    for (step = 1; step < 16; ++step) {
      knot = t[1] + (step / 16) * (t[2] - t[1]);
      for (k = 0; k < 3; ++k) A[k] = p[k] +
                ((knot - t[k]) / (t[k + 1] - t[k])) * (p[k + 1] - p[k]);
      for (k = 0; k < 2; ++k) B[k] = A[k] +
                ((knot - t[k]) / (t[k + 2] - t[k])) * (A[k + 1] - A[k]);
      upsampled_spline.push_back(B[0] +
                ((knot - t[1]) / (t[2] - t[1])) * (B[1] - B[0])));
    }
  }
  upsampled_spline.push_back(S[S.size() - 1]);
}
current = upsampled_spline[0]; next = 0;
all_samples.push_back({.point = current, .arclength = 1});
while (next < upsampled_spline.size()) {
  previous = current; arclength = 0.0;
  while (true) {
    if (next == upsampled_spline.size()) {
      all_samples.push_back({.point = previous, .arclength = arclength});
      break;
    }
    arclength_to_next = sqrt(pow(upsampled_spline[next].x - previous.x, 2) +
                             pow(upsampled_spline[next].y - previous.y, 2));
    if (arclength + arclength_to_next >= 1) {
      current = previous + ((1 - arclength) / arclength_to_next)
                         * (upsampled_spline[next] - previous);
      all_samples.push_back({.point = current, .arclength = 1});
      break;
    }
    arclength += arclength_to_next; previous = upsampled_spline[next];
    ++next;
  }
}
arclength = all_samples.size() - 2 + all_samples.back().arclength;
for (i = 0; i < all_samples.size(); ++i) {
  arclength_from_start = min(1.0, i / arclength);
  p = all_samples[i].point;
  d = all_samples[i].arclength;
  Sigma = ContinuousIDCT(dctSigma, 31 * arclength_from_start);

  /* for each channel C in {X, Y, B} */ {
    value = ContinuousIDCT(dctC, 31 * arclength_from_start);
    for (x = 0; x < width; ++x)
      for (y = 0; y < height; ++y) {
        dx = x - p.x; dy = y - p.y;
        distance = sqrt(dx * dx + dy * dy);
        factor = erf((distance / 2 + sqrt(1 / 8)) / Sigma)
               - erf((distance / 2 - sqrt(1 / 8)) / Sigma);
        /* the sample at column x and row y of channel C */ +=
          value * d * Sigma * factor * factor / 4;
      }
  }
}
```

NOTE In practical implementations, the range of the inner loop can be significantly reduced to a small square centered around `(p.x, p.y)` and with dimensions proportional to `Sigma` and `value`, since the amplitude of the sample updates becomes negligible for large enough values of `distance`.

K.5 Noise

K.5.1 Noise synthesis parameters

If the **kNoise** flag in `frame_header` (F.2) is not set, this subclause is skipped for that frame.

The 8 LUT values representing the noise synthesis parameters at different intensity levels are decoded sequentially as specified by the following code:

```
for (i = 0; i < 8; i++) lut[i] = u(10) / (1 << 10);
```

K.5.2 Noise rendering

If the **kNoise** flag in `frame_header` is not set, this subclause is skipped for that frame. Otherwise, noise is added to each group after rendering splines (K.4) as described in this subclause.

NOTE 1 Fully-specified noise generation has the advantage of allowing encoders to compensate for the noise that a decoder will add. The latitude (maximum deviation) between decoders is smaller than the largest magnitude of the generated noise.

The decoder generates pseudorandom channels of the same size as a group, modulates them, and adds their samples to the corresponding sample of the group.

Three pseudorandom channels denoted `RR`, `RG`, `RB` are generated in left to right order as follows. Rows in a channel are generated in top to bottom order. Each row is generated as follows.

For every segment of up to 16 samples, from left to right, the decoder generates an array of eight pseudorandom variables `batch[i]`, where i is [0, 8). Given arrays `s0[i]` and `s1[i]`, the decoder computes `batch[i]` as specified by the following code, where `s0_` and `s1_` are temporary variables.

```
s1_ = s0[i];
s0_ = s1[i];
batch[i] = (s1[i] + s0[i]) Umod (1<<64);
s0[i] = s0_;
s1_ ^= (s1_ << 23) Umod (1<<64);
s1[i] = s1_ ^ s0_ ^ (s1_ >> 18) ^ (s0_ >> 5);
```

`s0[i]` and `s1[i]` constitute the internal state of a XorShift128Plus generator, and their updated values are used to compute the next `batch[i]`. Before generating random numbers for a group, the decoder (re) initializes `s0` and `s1` as specified by the following code. Let `seed0` be equal to `(vis_frame_idx << 32) + invis_frame_idx` where `vis_frame_idx` is the number of 'visible' frames decoded so far (frames with `frame_type` equal to **kRegularFrame** or **kSkipProgressive** and either `duration > 0` or `is_last == true`, including the current frame if it is visible), and `invis_frame_idx` is the number of 'invisible' frames (frames with `frame_type` equal to **kLFFrame** or **kReferenceOnly**, or with `duration == 0` and `is_last == false`) since the previous visible frame (including the current frame if it is invisible). Let `seed1` be equal to `(x0 << 32) + y0` where `(x0, y0)` are the coordinates of the top-left pixel of the group within the frame.

```
SplitMix64(z) {
  z = ((z ^ (z >> 30)) * 0xBF58476D1CE4E5B9) Umod 1<<64;
  z = ((z ^ (z >> 27)) * 0x94D049BB133111EB) Umod 1<<64;
  return z ^ (z >> 31);
}
s0[0] = SplitMix64((seed0 + 0x9E3779B97F4A7C15) Umod 1<<64);
s1[0] = SplitMix64((seed1 + 0x9E3779B97F4A7C15) Umod 1<<64);
for (i = 1; i < 8; ++i) {
  s0[i] = SplitMix64(s0[i - 1]);
  s1[i] = SplitMix64(s1[i - 1]);
}
```

The resulting `batch[i]` is interpreted as a 16-element array of 32-bit values `bits[]`, where `bits[0]` are the lower 32 bits of `batch[0]`, `bits[1]` are the upper 32 bits of `batch[0]`, `bits[2]` are the lower 32 bits of `batch[1]`, and so forth. The next 16 (or the number of remaining samples in the row, whichever is less) samples `s[j]` are generated as specified by the following code:

```
s[j] = InterpretAsF32((bits[j] >> 9) | 0x3F800000);
```

After all rows are thus initialized, each channel is (in its totality, not in a per-group way) convolved by the following Laplacian-like 5×5 kernel:

0.16	0.16	0.16	0.16	0.16
0.16	0.16	0.16	0.16	0.16
0.16	0.16	−3.84	0.16	0.16
0.16	0.16	0.16	0.16	0.16
0.16	0.16	0.16	0.16	0.16

NOTE 2 This kernel corresponds to $-4\times(Id-Bk)$, where Bk is the box kernel.

The convolution potentially references input pixels with coordinates `(cx, cy)` outside the valid bounds. Every such access is redirected to coordinates `Mirror(cx, cy)` (5.2).

The following procedure is applied to each pixel.

`AR`, `AG`, `AB` denote the corresponding sample from each of the preceding pseudorandom channels `RR`, `RG`, `RB`, further scaled by multiplication with 0.22.

The strength of the noise is computed as follows. Given the `X` and `Y` samples of the input pixel, let `InR = (Y + X) / 2` and `InG = (Y - X) / 2`. Let `InscaledR = max(0, InR * 6)` and `InscaledG = max(0, InG * 6)`. These values are split into integer parts `in_intR = floor(InscaledR)` and `in_intG = floor(InscaledG)` and fractional parts `in_fracR = InscaledR - in_intR` and `in_fracG = InscaledG - in_intG`. If the value `InScaledR` or `InScaledG` is 7 or more, the corresponding integer and fractional parts are clamped back to 6 and 1 respectively.

Using the LUT values decoded as described in K.5.1, the strengths `SR` and `SG` are defined as `SR = LUT[in_intR] * (1 - in_fracR) + LUT[in_intR + 1] * in_fracR` and `SG = LUT[in_intG] * (1 - in_fracG) + LUT[in_intG + 1] * in_fracG`. Each strength is clamped between 0 and 1.

Using the additive noise `AR`, `AG`, `AB` and the strength `SR` and `SG`, the X, Y and B samples of each input pixel are modulated as specified by the following code, in which `base_correlation_x` and `base_correlation_b` are defined in I.2.3.

```
NR = 1/128 * AR * SR + 127/128 * AB * SR;
NG = 1/128 * AG * SG + 127/128 * AB * SG;
X += base_correlation_x * (NR + NG) + NR - NG;
Y += NR + NG;
B += base_correlation_b * (NR + NG);
```

Annex L
(normative)

Colour transforms

L.1 General

The decoded samples are in the colour space specified in metadata. In the case of `metadata.xyb_encoded == true`, this is an absolute colour space and the samples are transformed as defined in L.2. In the case of `metadata.xyb_encoded == false`, the samples are interpreted in the colour encoding indicated by `metadata.colour_encoding`. In the case of `frame_header.do_YCbCr`, the samples are converted to RGB as in L.3.

Modular modes may internally use additional decorrelating colour transforms which are reversed as specified in H.6.3 and H.6.4. Extra channels are rendered as specified in L.4.

L.2 XYB

L.2.1 OpsinInverseMatrix

OpsinInverseMatrix, defined in Table L.1, specifies parameters related to the inverse XYB colour transform.

Table L.1 — OpsinInverseMatrix bundle

condition	type	default	name
	Bool()	`true`	`all_default`
`!all_default`	F16()	11.031566901960783	`inv_mat00`
`!all_default`	F16()	−9.866943921568629	`inv_mat01`
`!all_default`	F16()	−0.16462299647058826	`inv_mat02`
`!all_default`	F16()	−3.254147380392157	`inv_mat10`
`!all_default`	F16()	4.418770392156863	`inv_mat11`
`!all_default`	F16()	−0.16462299647058826	`inv_mat12`
`!all_default`	F16()	−3.6588512862745097	`inv_mat20`
`!all_default`	F16()	2.7129230470588235	`inv_mat21`
`!all_default`	F16()	1.9459282392156863	`inv_mat22`
`!all_default`	F16()	−0.0037930732552754493	`opsin_bias0`
`!all_default`	F16()	−0.0037930732552754493	`opsin_bias1`
`!all_default`	F16()	−0.0037930732552754493	`opsin_bias2`
`!all_default`	F16()	1−0.05465007330715401	`quant_bias0`
`!all_default`	F16()	1−0.07005449891748593	`quant_bias1`
`!all_default`	F16()	1−0.049935103337343655	`quant_bias2`
`!all_default`	F16()	0.145	`quant_bias_numerator`

L.2.2 Inverse XYB transform

If `frame_header.encoding == ` **kModular**, the integer sample values `Y'`, `X'`, `B'` (the values in the first three channels) are first converted to `X`, `Y`, `B` samples as follows: `X = X' * m_x_lf_unscaled`, `Y = Y' * m_y_lf_unscaled`, and `B = (B' + Y') * m_b_lf_unscaled`.

X, Y, B samples are converted to an RGB colour encoding as specified in this subclause, in which `oim` denotes `metadata.opsin_inverse_matrix`. The colour space basis is changed and the gamma compression undone as specified by the following code:

```
Lgamma = Y + X;
Mgamma = Y - X;
Sgamma = B;
itscale = 255 / metadata.tone_mapping.intensity_target;
Lmix = (pow(Lgamma - cbrt(oim.opsin_bias0),3) + oim.opsin_bias0) * itscale;
Mmix = (pow(Mgamma - cbrt(oim.opsin_bias1),3) + oim.opsin_bias1) * itscale;
Smix = (pow(Sgamma - cbrt(oim.opsin_bias2),3) + oim.opsin_bias2) * itscale;
R = oim.inv_mat00 * Lmix + oim.inv_mat01 * Mmix + oim.inv_mat02 * Smix;
G = oim.inv_mat10 * Lmix + oim.inv_mat11 * Mmix + oim.inv_mat12 * Smix;
B = oim.inv_mat20 * Lmix + oim.inv_mat21 * Mmix + oim.inv_mat22 * Smix;
```

The resulting RGB samples correspond to sRGB primaries and a D65 white point, and the transfer function is linear. The resulting R, G and B samples are display-referred and are such that 1.0 represents an intensity of `intensity_target` cd/m^2 (nits).

NOTE The decoder can choose different primaries or white point by first multiplying `inv_mat` by a matrix converting from sRGB to the desired primaries and/or from D65 to the desired white point. The decoder can also apply a different transfer function. In this way, it can skip conversion to the colour space described in the image metadata, and directly convert to the display space.

In case the image has an alpha channel, spot colours, or frames that are stored as a reference frame with `save_before_ct == false`, and the image metadata does not describe a CMYK colour space, then the decoder converts the resulting RGB samples to the colour space described in the image metadata.

L.3 YCbCr

This transformation operates on the first three channels. These channels have the same dimensions. For every pixel position, the values (Cb, Y, Cr) are replaced by (R, G, B) values, as specified by the following code:

```
R = Y + 128.0 / 255 + 1.402 * Cr;
G = Y + 128.0 / 255 - 0.344136 * Cb - 0.714136 * Cr;
B = Y + 128.0 / 255 + 1.772 * Cb;
```

L.4 Extra channel rendering

Some extra channels are suitable for rendering into the main image. Decoders that wish to render an output image (instead of returning separate extra channels to the application), do so as follows.

At the final step in the decode process (when the inverse colour transform has been applied), the decoded image is in the colour space defined by the image metadata. In this subclause, `info[n]` denotes `metadata.ec_info[n]`.

The decoder then renders the extra channel with index n (in ascending order of n) as follows. First the channel is upsampled horizontally and vertically by a factor `f = ec_upsampling[n] << info[n].dim_shift`, using the upsampling described in K.2. If `f > 8`, then first an 8x upsampling is applied, followed by an upsampling by a factor of `f Idiv 8`. Note that `f` is at most 64. Then:

— If `info[n].type == `**kSpotColour** : For every (R, G, B) sample in the image and corresponding sample S in `extra_channel[n]` :

```
mix = S * info[n].solidity;
R = mix * info[n].red + (1 - mix) * R;
G = mix * info[n].green + (1 - mix) * G;
B = mix * info[n].blue + (1 - mix) * B;
```

— If `info[n].type == `**kAlpha**: This extra channel defines an alpha channel for the image. If there are multiple extra channels of type **kAlpha**, the first one (with the lowest index) contains the main alpha channel to use for blending. Other extra alpha channels can still be decoded, but are not considered the main alpha channel. The main alpha channel can be used for blending of animation frames as specified in F.2. The `alpha_associated` field, as specified in D.3.6, of the main alpha channel indicates whether the colour samples, as well as samples from other extra channels, are to be interpreted as having been premultiplied by their opacity.

— Else, nothing is done.

Annex M
(normative)

Profiles and levels

To promote interoperability, a single profile, named “Main” profile, is defined. This profile is intended for use (among others) in mobile phones, web browsers, and image editors. It includes all of the coding tools in this document.

The Main profile has two levels. Level 5 is suitable for end-user image delivery, including web browsers and mobile apps. Level 10 corresponds to a broad range of use cases such as image authoring workflows, print, scientific applications, satellite imagery, etc.

Levels are defined in such a way that if a decoder supports level N, it also supports lower levels.

Unless signalled otherwise, a JPEG XL codestream is assumed to be conforming to the Main profile, level 5.

The levels are defined in Table M.1.

Table M.1 — Levels in the Main Profiles

Parameter	Level 5	Level 10
Maximum `width * height` (D.2, F.2)	`1 << 28`	`1 << 40`
Maximum `width` (D.2, F.2)	`1 << 18`	`1 << 30`
Maximum `height` (D.2, F.2)	`1 << 18`	`1 << 30`
Maximum ICC `output_size` (E.4.2)	`1 << 22`	`1 << 28`
Maximum `bits_per_sample` (D.3, D.3.6)	16	32
Extra channel types (Table D.9)	any type except **kBlack**	any type
Maximum `num_extra` (Table D.3)	4	256
Maximum `num_splines` (K.4.1)	`min(1 << 24, fwidth * fheight / 4)`	
Maximum `total_num_control_points` (K.4.1)	`min(1 << 20, fwidth * fheight / 2)`	
Maximum `total_estimated_area_reached` (K.4.1)	`min(8 * fwidth * fheight + (1 << 25), 1 << 30)`	`min(1024 * fwidth * fheight + (1 << 32), 1 << 42)`
Maximum `num_patches` (K.3.1)	`min(1 << 24, fwidth * fheight / 16)`	
`modular_16bit_buffers` (Table D.3)	`true`	`true` or `false`
Maximum `nb_transforms` (H.2)	8	512
Maximum `nb_channels_tr`	256	`1 << 16`
Maximum `max_tree_depth` (H.4.2)	64	2048
Maximum global MA tree nodes (G.1.3, H.4.2)	`min(1 << 22, 1024 + fwidth * fheight * nb_channels / 16)`	
Maximum local MA tree nodes (G.2.3, G.4.2, H.4.2)	`min(1 << 20, 1024 + nb_local_samples)`	
Maximum total pixels in consecutive zero-duration frames	`1 << 28`	no limit
Minimum non-zero frame duration	1/120 second	no limit

The above parameters are defined as follows:

— `width` and `height` are the largest dimensions (horizontally and vertically, respectively) of the image and any of its frames, in pixels;

— `fwidth` and `fheight` are the dimensions of the current frame;

— `output_size` is the size in bytes of the ICC profile (E.4);

— `bits_per_sample` is the largest value of `bits_per_sample` (Table D.7) in any of the channels;

— `num_splines` and `num_patches` are the number of splines and patches (respectively) which are overlaid on a frame (Annex K);

— `total_num_control_points` is the sum of the values of `num_control_points` (as defined in K.4.1) for all splines in the current frame

— `total_estimated_area_reached` is the sum of the values of `estimated_area_reached` (as defined in K.4.1) for all splines in the current frame.

— `nb_transforms` is the largest number of modular transforms for any group (including transforms in the GlobalModular section);

— `nb_channels` is the total number of channels (colour plus extra channels);

— `nb_channels_tr` is the number of modular channels that is derived from the channel initialization and the series of transforms, as described in H.2;

— `max_tree_depth` is the largest tree depth (maximum distance from root to any leaf node) of the MA trees that are used in modular encoding, cf. H.4.2;

— `nb_local_samples` is the total number of samples to be decoded in any given modular sub-bitstream.

Minimum non-zero frame duration is to be interpreted as follows: if the frame duration is above this threshold, the decoder shall make the best effort to respect the value; if the frame duration is below this threshold (but not zero), the decoder is allowed to behave as if the frame duration is equal to this threshold.

Annex N
(normative)

Extensions

The extensions field in some header bundles enables backward- and forward-compatible codestream extensions as described in B.3.

Extensions for ImageMetadata, FrameHeader, and RestorationFilter are not currently defined, so the decoder reads and ignores any extension bits specified there.

Annex O
(informative)

Encoder overview

O.1 Overview

An encoder is an embodiment of the encoding process. This document does not specify an encoding process, and any encoding process is acceptable as long as the codestream is as specified in this document. This Annex gives an overview of the steps that might be involved.

The encoder can choose the FrameEncoding based on input type and acceptable loss: **kModular** for non-photographic inputs or for (mathematically) lossless encoding; and **kVarDCT** for lossy or for lossless JPEG image recompression – existing JPEG images can be represented using the **kVarDCT** encoding using only `DctSelect` type DCT8×8 and skipping most of the coding tools defined in this document.

For **kModular**, the encoder can apply a palette transform if the image contains few colours, a colour transform to decorrelate the channels, and the Haar-wavelet-like squeeze transform to obtain a multiresolution representation. The latter recursively replaces each channel with two channels: one downsampled in horizontal or vertical direction, and the other containing residual values with respect to a predictor. The downsampling averages two pixels, rounding up if the left/top one is greater, otherwise down (this prevents bias).

For **kVarDCT**, the encoder first transforms pixels into the XYB colour space. It can then perform the inverse operation (sharpening) to compensate for the blurring caused by the decoder restoration filter's convolution.

To partition the image into varblocks, the encoder can compute a DCT for naturally-aligned 32 × 32, 16 × 16, and 8 × 8 rectangles, and then choose the varblock sizes such that entropy of their coefficients is minimized.

After computing varblock DCTs, encoders can compute a LF image by taking LLF and computing the IDCT of size N / 8 of them to create a low resolution image (conceptually a 1:8 downscale of the original image).

O.2 Entropy encoding

The ANS encoder keeps an internal 32-bit state. Upon creation of the encoder (immediately before writing the first symbol from a new ANS stream), state is initialized with `0x130000`. The encoder writes symbols in reverse order (LIFO). It encodes a symbol belonging to a given distribution D as specified by the following code:

```
if ((state >> 20) >= D[symbol]) {
  /* write state & 0xFFFF as a u(16) */
  state >>= 16;
}
/* k is chosen so AliasMapping(D, k) = (symbol, state Umod D[symbol]) */
state = ((state Idiv D[symbol]) << 12) + k;
```

O.3 Forward transforms

O.3.1 Quantization

Dequantization involves multiplying coefficients with a function of quantization tables and a scaling factor (global for LF and per-block for HF). Conversely, quantization in the encoder involves division.

The encoder can search for the X/BFromY that result in the largest number of quantized zero residuals, and an adaptive quantization map that minimizes a (perceptual) distance metric between the input and the expected decoder output.

The X, Y, B predictor residuals are quantized by rounding the value after multiplying by the quantization matrix times the adaptive quantization entry for the 8 × 8 block.

O.3.2 Squeeze

The horizontal forward squeeze step takes one input channels and replaces it with two output channels; the vertical squeeze step is similar. The input channel has dimensions `W` × `H`; the resulting output channels have dimensions `W1` × `H` and `W2` × `H`, where `W1 = ceil(W/2)` and `W2 = floor(W/2)`. The output channels are constructed as specified by the following code:

```
horiz_fsqueeze(input, output_1, output_2) {
  for (y = 0; y < H; y++) {
    for (x = 0; x < W2; x++) {
      A = input(2 * x, y);
      B = input(2 * x + 1, y);
      avg = (A + B + (A > B ? 1 : 0)) >> 1;
      output_1(x, y) = avg;
      diff = A - B;
      next_avg = avg;
      if (x + 1 < W1) {
        next_avg = (input(2 * x + 2, y) + input(2 * x + 3, y) +
          (input(2 * x + 2, y) > input(2 * x + 3, y) ? 1 : 0) >> 1;
      } else if (W Umod 2 == 1) next_avg = input(2 * x + 2, y);
      left = (x > 0 ? input(2 * x - 1, y) : avg);
      residu = diff - tendency(left, avg, next_avg);
      output_2(x, y) = residu;
    }
    if (W1 > W2) output_1(W2) = input(2 * W2);
  }
}
```

O.3.3 XYB

Starting with RGB samples with the sRGB primaries and a linear transfer curve, the following code describes the forward XYB transform (corresponding to the default values for `metadata.opsin_inverse_matrix`).

```
bias = -0.00379307325527544933;
Lmix = 0.3 * R + 0.622 * G + 0.078 * B - bias;
Mmix = 0.23 * R + 0.692 * G + 0.078 * B - bias;
Smix =  0.24342268924547819 * R + 0.20476744424496821 * G
      + 0.55180986650955360 * B - bias;
Lgamma = cbrt(Lmix) + cbrt(bias);
Mgamma = cbrt(Mmix) + cbrt(bias);
Sgamma = cbrt(Smix) + cbrt(bias);
X = (Lgamma - Mgamma) / 2;
Y = (Lgamma + Mgamma) / 2;
B = Sgamma;
```

O.3.4 RCT

This transformation operates on three channels starting with `begin_c`. These channels have the same dimensions. For every pixel position in these channels, the values `(V[0], V[1], V[2])` are replaced by values `(A, B, C)`, as specified by the following code:

```
permutation = rct_type Idiv 7;
type = rct_type Umod 7;
second = type >> 1;
third = (type & 1);
D = V[permutation Umod 3];
E = V[permutation+1+(permutation Idiv 3) Umod 3];
F = V[permutation+2-(permutation Idiv 3) Umod 3];

if (type == 6) {        // YCoCg
  B = D - F;
  tmp = F + (B >> 1);
  C = E - tmp;
  A = tmp + (C >> 1);
} else {
  if (second == 1) E = E - D;
  if (second == 2) E = E - ((D + F) >> 1);
  if (third == 1) F = F - D;
  A = D; B = E; C = F;
}
```

EXAMPLE If `rct_type == 10`, then pixels `(R, G, B)` are replaced by `(G, B - G, G, R - G)`.

Bibliography

[1] ISO/IEC 14882, *Programming languages — C++*

[2] ISO/IEC 23091-2, *Information technology — Coding-independent code points — Part 2: Video*

[3] JPEG COMMITTEE *JPEG XL website*. https://jpeg.org/jpegxl

ICS 35.040.30
Price based on 91 pages